Python

PARA TODOS

Raúl González Duque



Python para todos es un tutorial básico del lenguaje de programación Python escrito por Raúl González Duque y distribuido gratuitamente bajo licencia Creative Commons.

Adecuado para cualquier nivel, *Python para todos* repasa aspectos esenciales de este lenguaje: tipos básicos, control de flujo, funciones, orientación a objetos, programación funcional, excepciones, entrada/salida, etcétera.

La introducción te permitirá conocer las características principales de Python, cómo instalarlo y por qué hacerlo.

Lectulandia

Raúl González Duque

Python para todos

ePub r1.0
Titivillus 10.12.2017

Título original: *Python para todos*
Raúl González Duque, 2008
Imagen de portada: Ian Chien

Editor digital: Titivillus
ePub base r1.2

Python para todos
por Raúl González Duque

Puede descargar la versión más reciente de este libro gratuitamente en la web http://mundogeek.net/tutorial-python/

INTRODUCCIÓN

¿Qué es Python?

Python es un lenguaje de programación creado por Guido van Rossum a principios de los años 90 cuyo nombre está inspirado en el grupo de cómicos ingleses "Monty Python". Es un lenguaje similar a Perl, pero con una sintaxis muy limpia y que favorece un código legible.

Se trata de un lenguaje interpretado o de script, con tipado dinámico, fuertemente tipado, multiplataforma y orientado a objetos.

Lenguaje interpretado o de script

Un lenguaje interpretado o de script es aquel que se ejecuta utilizando un programa intermedio llamado intérprete, en lugar de compilar el código a lenguaje máquina que pueda comprender y ejecutar directamente una computadora (lenguajes compilados).

La ventaja de los lenguajes compilados es que su ejecución es más rápida. Sin embargo los lenguajes interpretados son más flexibles y más portables.

Python tiene, no obstante, muchas de las características de los lenguajes compilados, por lo que se podría decir que es semi interpretado. En Python, como en Java y muchos otros lenguajes, el código fuente se traduce a un pseudo código máquina intermedio llamado bytecode la primera vez que se ejecuta, generando archivos .pyc o .pyo (bytecode optimizado), que son los que se ejecutarán en sucesivas ocasiones.

Tipado dinámico

La característica de tipado dinámico se refiere a que no es necesario declarar el tipo de dato que va a contener una determinada variable, sino que su tipo se determinará en tiempo de ejecución según el tipo del valor al que se asigne, y el tipo de esta variable puede cambiar si se le asigna un valor de otro tipo.

Fuertemente tipado

No se permite tratar a una variable como si fuera de un tipo distinto al que tiene, es necesario convertir de forma explícita dicha variable al nuevo tipo previamente. Por ejemplo, si tenemos una variable que contiene un texto (variable de tipo cadena o string) no podremos tratarla como un número (sumar la cadena "9" y el número 8). En otros lenguajes el tipo de la variable cambiaría para adaptarse al comportamiento esperado, aunque esto es más propenso a errores.

Multiplataforma

El intérprete de Python está disponible en multitud de plataformas (UNIX, Solaris,

Linux, DOS, Windows, OS/2, Mac OS, etc.) por lo que si no utilizamos librerías específicas de cada plataforma nuestro programa podrá correr en todos estos sistemas sin grandes cambios.

Orientado a objetos

La orientación a objetos es un paradigma de programación en el que los conceptos del mundo real relevantes para nuestro problema se trasladan a clases y objetos en nuestro programa. La ejecución del programa consiste en una serie de interacciones entre los objetos.

Python también permite la programación imperativa, programación funcional y programación orientada a aspectos.

¿Por qué Python?

Python es un lenguaje que todo el mundo debería conocer. Su sintaxis simple, clara y sencilla; el tipado dinámico, el gestor de memoria, la gran cantidad de librerías disponibles y la potencia del lenguaje, entre otros, hacen que desarrollar una aplicación en Python sea sencillo, muy rápido y, lo que es más importante, divertido.

La sintaxis de Python es tan sencilla y cercana al lenguaje natural que los programas elaborados en Python parecen pseudocódigo. Por este motivo se trata además de uno de los mejores lenguajes para comenzar a programar.

Python no es adecuado sin embargo para la programación de bajo nivel o para aplicaciones en las que el rendimiento sea crítico.

Algunos casos de éxito en el uso de Python son Google, Yahoo, la NASA, Industrias Light & Magic, y todas las distribuciones Linux, en las que Python cada vez representa un tanto por ciento mayor de los programas disponibles.

Instalación de Python

Existen varias implementaciones distintas de Python: CPython, Jython, IronPython, PyPy, etc.

CPython es la más utilizada, la más rápida y la más madura. Cuando la gente habla de Python normalmente se refiere a esta implementación.

En este caso tanto el intérprete como los módulos están escritos en C.

Jython es la implementación en Java de Python, mientras que IronPython es su contrapartida en C# (.NET). Su interés estriba en que utilizando estas implementaciones se pueden utilizar todas las librerías disponibles para los programadores de Java y .NET.

PyPy, por último, como habréis adivinado por el nombre, se trata de una implementación en Python de Python.

CPython está instalado por defecto en la mayor parte de las distribuciones Linux y en las últimas versiones de Mac OS. Para comprobar si está instalado abre una terminal y escribe python. Si está instalado se iniciará la consola interactiva de Python y obtendremos algo parecido a lo siguiente:

```
Python 2.5.1 (r251:54863, May 2 2007, 16:56:35)
[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2
Type “help”, “copyright”, “credits” or “license” for more information.
>>>
```

La primera línea nos indica la versión de Python que tenemos instalada. Al final podemos ver el prompt (>>>) que nos indica que el intérprete está esperando código del usuario. Podemos salir escribiendo `exit()`, o pulsando Control + D.

Si no te muestra algo parecido no te preocupes, instalar Python es muy sencillo. Puedes descargar la versión correspondiente a tu sistema operativo desde la web de Python, en *http://www.python.org/download/*.

Existen instaladores para Windows y Mac OS. Si utilizas Linux es muy probable que puedas instalarlo usando la herramienta de gestión de paquetes de tu distribución, aunque también podemos descargar la aplicación compilada desde la web de Python.

Herramientas básicas

Existen dos formas de ejecutar código Python. Podemos escribir líneas de código en el intérprete y obtener una respuesta del intérprete para cada línea (sesión interactiva) o bien podemos escribir el código de un programa en un archivo de texto y ejecutarlo.

A la hora de realizar una sesión interactiva os aconsejo instalar y utilizar iPython, en lugar de la consola interactiva de Python. Se puede encontrar en *http://ipython.scipy.org/*. iPython cuenta con características añadidas muy interesantes, como el autocompletado o el operador `?`. (para activar la característica de autocompletado en Windows es necesario instalar `PyReadline`, que puede descargarse desde *http://ipython.scipy.org/moin/PyReadline/Intro*)

La función de autocompletado se lanza pulsando el tabulador. Si escribimos `fi` y pulsamos Tab nos mostrará una lista de los objetos que comienzan con `fi` (`file`, `filter` y `finally`). Si escribimos `file.` y pulsamos Tab nos mostrará una lista de los métodos y propiedades del objeto `file`.

El operador `?` nos muestra información sobre los objetos. Se utiliza añadiendo el símbolo de interrogación al final del nombre del objeto del cual queremos más información. Por ejemplo:

```
In [3]: str?
Type: type
Base Class:
String Form:
Namespace: Python builtin
Docstring:
str(object) -> string

Return a nice string representation of the object.
If the argument is a string, the return value is the same object.
```

En el campo de IDEs y editores de código gratuitos PyDEV (*http://pydev.sourceforge.net/*) se alza como cabeza de serie. PyDEV es un plugin para Eclipse que permite utilizar este IDE multiplataforma para programar en Python. Cuenta con autocompletado de código (con información sobre cada elemento), resaltado de sintaxis, un depurador gráfico, resaltado de errores, explorador de clases, formateo del código, refactorización, etc. Sin duda es la opción más completa, sobre todo si instalamos las extensiones comerciales, aunque necesita de una cantidad importante de memoria y no es del todo estable.

Otras opciones gratuitas a considerar son SPE o Stani's Python Editor (*http://sourceforge.net/projects/spe/*), Eric (*http://die-offenbachs.de/eric/*), BOA Constructor (*http://boa-constructor.sourceforge.net/*) o incluso emacs o vim.

Si no te importa desembolsar algo de dinero, Komodo (*http://www.activestate.com/komodo_ide/*) y Wing IDE (*http://www.wingware.com/*)

son también muy buenas opciones, con montones de características interesantes, como PyDEV, pero mucho más estables y robustos. Además, si desarrollas software libre no comercial puedes contactar con Wing Ware y obtener, con un poco de suerte, una licencia gratuita para Wing IDE Professional :)

MI PRIMER PROGRAMA EN PYTHON

Como comentábamos en el capítulo anterior existen dos formas de ejecutar código Python, bien en una sesión interactiva (línea a línea) con el intérprete, o bien de la forma habitual, escribiendo el código en un archivo de código fuente y ejecutándolo.

El primer programa que vamos a escribir en Python es el clásico Hola Mundo, y en este lenguaje es tan simple como:

```
print "Hola Mundo"
```

Vamos a probarlo primero en el intérprete. Ejecuta python o ipython según tus preferencias, escribe la línea anterior y pulsa Enter. El intérprete responderá mostrando en la consola el texto `Hola Mundo`.

Vamos ahora a crear un archivo de texto con el código anterior, de forma que pudiéramos distribuir nuestro pequeño gran programa entre nuestros amigos. Abre tu editor de texto preferido o bien el IDE que hayas elegido y copia la línea anterior. Guárdalo como hola.py, por ejemplo.

Ejecutar este programa es tan sencillo como indicarle el nombre del archivo a ejecutar al intérprete de Python

```
python hola.py
```

pero vamos a ver cómo simplificarlo aún más.

Si utilizas Windows los archivos .py ya estarán asociados al intérprete de Python, por lo que basta hacer doble clic sobre el archivo para ejecutar el programa. Sin embargo como este programa no hace más que imprimir un texto en la consola, la ejecución es demasiado rápida para poder verlo siquiera. Para remediarlo, vamos a añadir una nueva línea que espere la entrada de datos por parte del usuario.

```
print "Hola Mundo"
raw_input()
```

De esta forma se mostrará una consola con el texto `Hola Mundo` hasta que pulsemos Enter.

Si utilizas Linux (u otro Unix) para conseguir este comportamiento, es decir, para que el sistema operativo abra el archivo .py con el intérprete adecuado, es necesario añadir una nueva línea al principio del archivo:

```
#!/usr/bin/python
print "Hola Mundo"
raw_input()
```

A esta línea se le conoce en el mundo Unix como *shebang*, *hashbang* o *sharpbang*. El par de caracteres `#!` indica al sistema operativo que dicho script se debe ejecutar utilizando el intérprete especificado a continuación. De esto se desprende,

evidentemente, que si esta no es la ruta en la que está instalado nuestro intérprete de Python, es necesario cambiarla.

Otra opción es utilizar el programa env (de environment, entorno) para preguntar al sistema por la ruta al intérprete de Python, de forma que nuestros usuarios no tengan ningún problema si se diera el caso de que el programa no estuviera instalado en dicha ruta:

```
#!/usr/bin/env python
print “Hola Mundo”
raw_input()
```

Por supuesto además de añadir el shebang, tendremos que dar permisos de ejecución al programa.

```
chmod +x hola.py
```

Y listo, si hacemos doble clic el programa se ejecutará, mostrando una consola con el texto `Hola Mundo`, como en el caso de Windows.

También podríamos correr el programa desde la consola como si tratara de un ejecutable cualquiera:

```
./hola.py
```

TIPOS BÁSICOS

En Python los tipos básicos se dividen en:

- Números, como pueden ser `3` (entero), `15.57` (de coma flotante) o `7 + 5j` (complejos)
- Cadenas de texto, como “`Hola Mundo`”
- Valores booleanos: `True` (cierto) y `False` (falso).

Vamos a crear un par de variables a modo de ejemplo. Una de tipo cadena y una de tipo entero:

```
# esto es una cadena
c = “Hola Mundo”

# y esto es un entero
e = 23

# podemos comprobarlo con la función type
type(c)
type(e)
```

Como veis en Python, a diferencia de muchos otros lenguajes, no se declara el tipo de la variable al crearla. En Java, por ejemplo, escribiríamos:

```
String c = “Hola Mundo”;
int e = 23;
```

Este pequeño ejemplo también nos ha servido para presentar los comentarios inline en Python: cadenas de texto que comienzan con el carácter `#` y que Python ignora totalmente. Hay más tipos de comentarios, de los que hablaremos más adelante.

Números

Como decíamos, en Python se pueden representar números enteros, reales y complejos.

Enteros

Los números enteros son aquellos números positivos o negativos que no tienen decimales (además del cero). En Python se pueden representar mediante el tipo `int` (de integer, entero) o el tipo `long` (largo). La única diferencia es que el tipo `long` permite almacenar números más grandes. Es aconsejable no utilizar el tipo `long` a menos que sea necesario, para no malgastar memoria.

El tipo `int` de Python se implementa a bajo nivel mediante un tipo `long` de C. Y dado que Python utiliza C por debajo, como C, y a diferencia de Java, el rango de los valores que puede representar depende de la plataforma.

En la mayor parte de las máquinas el `long` de C se almacena utilizando 32 bits, es decir, mediante el uso de una variable de tipo `int` de Python podemos almacenar números de -2^{31} a $2^{31}-1$, o lo que es lo mismo, de -2.147.483.648 a 2.147.483.647. En plataformas de 64 bits, el rango es de -9.223.372.036.854.775.808 hasta 9.223.372.036.854.775.807.

El tipo `long` de Python permite almacenar números de cualquier precisión, estando limitados solo por la memoria disponible en la máquina.

Al asignar un número a una variable esta pasará a tener tipo `int`, a menos que el número sea tan grande como para requerir el uso del tipo `long`.

```
# type(entero) devolvería int
entero = 23
```

También podemos indicar a Python que un número se almacene usando `long` añadiendo una L al final:

```
# type(entero) devolvería long
entero = 23L
```

El literal que se asigna a la variable también se puede expresar como un octal, anteponiendo un cero:

```
# 027 octal = 23 en base 10
entero = 027
```

o bien en hexadecimal, anteponiendo un `0x`:

```
# 0x17 hexadecimal = 23 en base 10
entero = 0x17
```

Reales

Los números reales son los que tienen decimales. En Python se expresan mediante el tipo `float`. En otros lenguajes de programación, como C, tenemos también el tipo `double`, similar a `float` pero de mayor precisión (`double` = doble precisión). Python, sin embargo, implementa su tipo `float` a bajo nivel mediante una variable de tipo `double` de C, es decir, utilizando 64 bits, luego en Python siempre se utiliza doble precisión, y en concreto se sigue el estándar IEEE 754: 1 bit para el signo, 11 para el exponente, y 52 para la mantisa. Esto significa que los valores que podemos representar van desde $\pm 2{,}2250738585072020 \times 10^{-308}$ hasta $\pm 1{,}7976931348623157 \times 10^{308}$.

La mayor parte de los lenguajes de programación siguen el mismo esquema para la representación interna. Pero como muchos sabréis esta tiene sus limitaciones, impuestas por el hardware. Por eso desde Python 2.4 contamos también con un nuevo tipo Decimal, para el caso de que se necesite representar fracciones de forma más precisa. Sin embargo este tipo está fuera del alcance de este tutorial, y sólo es necesario para el ámbito de la programación científica y otros relacionados. Para aplicaciones normales podéis utilizar el tipo `float` sin miedo, como ha venido haciéndose desde hace años, aunque teniendo en cuenta que los números en coma flotante no son precisos (ni en este ni en otros lenguajes de programación).

Para representar un número real en Python se escribe primero la parte entera, seguido de un punto y por último la parte decimal.

```
real = 0.2703
```

También se puede utilizar notación científica, y añadir una `e` (de exponente) para indicar un exponente en base 10. Por ejemplo:

```
real = 0.1e-3
```

sería equivalente a $0.1 \times 10^{-3} = 0.1 \times 0.001 = 0.0001$

Complejos

Los números complejos son aquellos que tienen parte imaginaria. Si no conocías de su existencia, es más que probable que nunca lo vayas a necesitar, por lo que puedes saltarte este apartado tranquilamente. De hecho la mayor parte de lenguajes de programación carecen de este tipo, aunque sea muy utilizado por ingenieros y científicos en general.

En el caso de que necesitéis utilizar números complejos, o simplemente tengáis curiosidad, os diré que este tipo, llamado `complex` en Python, también se almacena usando coma flotante, debido a que estos números son una extensión de los números reales. En concreto se almacena en una estructura de C, compuesta por dos variables

de tipo `double`, sirviendo una de ellas para almacenar la parte real y la otra para la parte imaginaria.

Los números complejos en Python se representan de la siguiente forma:

```
complejo = 2.1 + 7.8j
```

Operadores

Veamos ahora qué podemos hacer con nuestros números usando los operadores por defecto. Para operaciones más complejas podemos recurrir al módulo `math`.

Operadores aritméticos

Operador	Descripción	Ejemplo
`+`	Suma	`r = 3 + 2 # r es 5`
`-`	Resta	`r = 4 - 7 # r es -3`
`-`	Negación	`r = -7 # r es -7`
`*`	Multiplicación	`r = 2 * 6 # r es 12`
`**`	Exponente	`r = 2 ** 6 # r es 64`
`/`	División	`r = 3.5 / 2 # r es 1.75`
`//`	División entera	`r = 3.5 // 2 # r es 1.0`
`%`	Módulo	`r = 7 % 2 # r es 1`

Puede que tengáis dudas sobre cómo funciona el operador de módulo, y cuál es la diferencia entre división y división entera.

El operador de módulo no hace otra cosa que devolvernos el resto de la división entre los dos operandos. En el ejemplo, 7/2 sería 3, con 1 de resto, luego el módulo es 1.

La diferencia entre división y división entera no es otra que la que indica su nombre. En la división el resultado que se devuelve es un número real, mientras que en la división entera el resultado que se devuelve es solo la parte entera.

No obstante hay que tener en cuenta que si utilizamos dos operandos enteros, Python determinará que queremos que la variable resultado también sea un entero, por lo que el resultado de, por ejemplo, `3 / 2` y `3 // 2` sería el mismo: 1.

Si quisiéramos obtener los decimales necesitaríamos que al menos uno de los operandos fuera un número real, bien indicando los decimales

```
r = 3.0 / 2
```

o bien utilizando la función `float` (no es necesario que sepáis lo que significa el término función, ni que recordéis esta forma, lo veremos un poco más adelante):

```
r = float(3) / 2
```

Esto es así porque cuando se mezclan tipos de números, Python convierte todos los operandos al tipo más complejo de entre los tipos de los operandos.

Operadores a nivel de bit

Si no conocéis estos operadores es poco probable que vayáis a necesitarlos, por lo que podéis obviar esta parte. Si aún así tenéis curiosidad os diré que estos son operadores que actúan sobre las representaciones en binario de los operandos.

Por ejemplo, si veis una operación como `3 & 2`, lo que estáis viendo es un and bit a bit entre los números binarios 11 y 10 (las representaciones en binario de 3 y 2).

El operador *and* (`&`), del inglés "y", devuelve 1 si el primer bit operando es 1 **y** el segundo bit operando es 1. Se devuelve 0 en caso contrario.

El resultado de aplicar and bit a bit a 11 y 10 sería entonces el número binario 10, o lo que es lo mismo, 2 en decimal (el primer dígito es 1 para ambas cifras, mientras que el segundo es 1 sólo para una de ellas).

El operador *or* (`|`), del inglés "o", devuelve 1 si el primer operando es 1 **o** el segundo operando es 1. Para el resto de casos se devuelve 0.

El operador *xor* u or exclusivo (`^`) devuelve 1 si uno de los operandos es 1 y el otro no lo es.

El operador *not* (`~`), del inglés "no", sirve para negar uno a uno cada bit; es decir, si el operando es 0, cambia a 1 y si es 1, cambia a 0.

Por último los operadores de desplazamiento (<< y >>) sirven para desplazar los bits n posiciones hacia la izquierda o la derecha.

<table>
<tr><th>Operador</th><th>Descripción</th><th>Ejemplo</th></tr>
<tr><td>&</td><td>and</td><td><code>r = 3 & 2 # r es 2</code></td></tr>
<tr><td>|</td><td>or</td><td><code>r = 3 | 2 # r es 3</code></td></tr>
<tr><td>^</td><td>xor</td><td><code>r = 3 ^ 2 # r es 1</code></td></tr>
<tr><td>~</td><td>not</td><td><code>r = ~3 # r es -4</code></td></tr>
<tr><td><<</td><td>Desplazamiento izq.</td><td><code>r = 3 << 1 # r es 6</code></td></tr>
<tr><td>>></td><td>Desplazamiento der.</td><td><code>r = 3 >> 1 # r es 1</code></td></tr>
</table>

Cadenas

Las cadenas no son más que texto encerrado entre comillas simples (`‘cadena’`) o dobles (`“cadena”`). Dentro de las comillas se pueden añadir caracteres especiales escapándolos con `\`, como `\n`, el carácter de nueva línea, o `\t`, el de tabulación.

Una cadena puede estar precedida por el carácter `u` o el carácter `r`, los cuales indican, respectivamente, que se trata de una cadena que utiliza codificación Unicode y una cadena *raw* (del inglés, cruda). Las cadenas `raw` se distinguen de las normales en que los caracteres escapados mediante la barra invertida (\) no se sustituyen por sus contrapartidas. Esto es especialmente útil, por ejemplo, para las expresiones regulares, como veremos en el capítulo correspondiente.

```
unicode = u“äóè”
raw = r“\n”
```

También es posible encerrar una cadena entre triples comillas (simples o dobles). De esta forma podremos escribir el texto en varias líneas, y al imprimir la cadena, se respetarán los saltos de línea que introdujimos sin tener que recurrir al carácter `\n`, así como las comillas sin tener que escaparlas.

```
triple = “““primera linea
     esto se vera en otra linea”””
```

Las cadenas también admiten operadores como `+`, que funciona realizando una concatenación de las cadenas utilizadas como operandos y `*`, en la que se repite la cadena tantas veces como lo indique el número utilizado como segundo operando.

```
a = “uno”
b = “dos”

c = a + b # c es “unodos”
c = a * 3 # c es “unounouno”
```

Booleanos

Como decíamos al comienzo del capítulo una variable de tipo booleano sólo puede tener dos valores: `True` (cierto) y `False` (falso). Estos valores son especialmente importantes para las expresiones condicionales y los bucles, como veremos más adelante.

En realidad el tipo `bool` (el tipo de los booleanos) es una subclase del tipo `int`. Puede que esto no tenga mucho sentido para ti si no conoces los términos de la orientación a objetos, que veremos más adelante, aunque tampoco es nada importante.

Estos son los distintos tipos de operadores con los que podemos trabajar con valores booleanos, los llamados operadores lógicos o condicionales:

Operador	**Descripción**	**Ejemplo**
`and`	¿se cumple a y b?	`r = True and False # r es False`
`or`	¿se cumple a o b?	`r = True or False # r es True`
`not`	No a	`r = not True # r es False`

Los valores booleanos son además el resultado de expresiones que utilizan operadores relacionales (comparaciones entre valores):

Operador	**Descripción**	**Ejemplo**
`==`	¿son iguales a y b?	`r = 5 == 3 # r es False`
`!=`	¿son distintos a y b?	`r = 5 != 3 # r es True`
`<`	¿es a menor que b?	`r = 5 < 3 # r es False`
`>`	¿es a mayor que b?	`r = 5 > 3 # r es True`
`<=`	¿es a menor o igual que b?	`r = 5 <= 5 # r es True`
`>=`	¿es a mayor o igual que b?	`r = 5 >= 3 # r es True`

COLECCIONES

En el capítulo anterior vimos algunos tipos básicos, como los números, las cadenas de texto y los booleanos. En esta lección veremos algunos tipos de colecciones de datos: listas, tuplas y diccionarios.

Listas

La lista es un tipo de colección ordenada. Sería equivalente a lo que en otros lenguajes se conoce por arrays, o vectores.

Las listas pueden contener cualquier tipo de dato: números, cadenas, booleanos, … y también listas.

Crear una lista es tan sencillo como indicar entre corchetes, y separados por comas, los valores que queremos incluir en la lista:

```
l = [22, True, “una lista”, [1, 2]]
```

Podemos acceder a cada uno de los elementos de la lista escribiendo el nombre de la lista e indicando el índice del elemento entre corchetes.

Ten en cuenta sin embargo que el índice del primer elemento de la lista es 0, y no 1:

```
l = [11, False]
mi_var = l[0] # mi_var vale 11
```

Si queremos acceder a un elemento de una lista incluida dentro de otra lista tendremos que utilizar dos veces este operador, primero para indicar a qué posición de la lista exterior queremos acceder, y el segundo para seleccionar el elemento de la lista interior:

```
l = [“una lista”, [1, 2]]
mi_var = l[1][0] # mi_var vale 1
```

También podemos utilizar este operador para modificar un elemento de la lista si lo colocamos en la parte izquierda de una asignación:

```
l = [22, True]
l[0] = 99 # Con esto l valdrá [99, True]
```

El uso de los corchetes para acceder y modificar los elementos de una lista es común en muchos lenguajes, pero Python nos depara varias sorpresas muy agradables.

Una curiosidad sobre el operador `[]` de Python es que podemos utilizar también números negativos. Si se utiliza un número negativo como índice, esto se traduce en que el índice empieza a contar desde el final, hacia la izquierda; es decir, con `[-1]` accederíamos al último elemento de la lista, con `[-2]` al penúltimo, con `[-3]`, al antepenúltimo, y así sucesivamente.

Otra cosa inusual es lo que en Python se conoce como *slicing* o particionado, y que consiste en ampliar este mecanismo para permitir seleccionar porciones de la lista. Si en lugar de un número escribimos dos números `inicio` y `fin` separados por dos puntos (`inicio:fin`) Python interpretará que queremos una lista que vaya desde la posición `inicio` a la posición `fin`, sin incluir este último. Si escribimos tres números (`inicio:fin:salto`) en lugar de dos, el tercero se utiliza para determinar cada

cuantas posiciones añadir un elemento a la lista.

```
l = [99, True, “una lista”, [1, 2]]
mi_var = l[0:2] # mi_var vale [99, True]
mi_var = l[0:4:2] # mi_var vale [99, “una lista”]
```

Los números negativos también se pueden utilizar en un slicing, con el mismo comportamiento que se comentó anteriormente.

Hay que mencionar así mismo que no es necesario indicar el principio y el final del slicing, sino que, si estos se omiten, se usarán por defecto las posiciones de inicio y fin de la lista, respectivamente:

```
l = [99, True, “una lista”]
mi_var = l[1:] # mi_var vale [True, “una lista”]
mi_var = l[:2] # mi_var vale [99, True]
mi_var = l[:] # mi_var vale [99, True, “una lista”]
mi_var = l[::2] # mi_var vale [99, “una lista”]
```

También podemos utilizar este mecanismo para modificar la lista:

```
l = [99, True, “una lista”, [1, 2]]
l[0:2] = [0, 1] # l vale [0, 1, “una lista”, [1, 2]]
```

pudiendo incluso modificar el tamaño de la lista si la lista de la parte derecha de la asignación tiene un tamaño menor o mayor que el de la selección de la parte izquierda de la asignación:

```
l[0:2] = [False] # l vale [False, “una lista”, [1, 2]]
```

En todo caso las listas ofrecen mecanismos más cómodos para ser modificadas a través de las funciones de la clase correspondiente, aunque no veremos estos mecanismos hasta más adelante, después de explicar lo que son las clases, los objetos y las funciones.

Tuplas

Todo lo que hemos explicado sobre las listas se aplica también a las tuplas, a excepción de la forma de definirla, para lo que se utilizan paréntesis en lugar de corchetes.

```
t = (1, 2, True, “python”)
```

En realidad el constructor de la tupla es la coma, no el paréntesis, pero el intérprete muestra los paréntesis, y nosotros deberíamos utilizarlos, por claridad.

```
>>> t = 1, 2, 3
>>> type(t)
type “tuple”
```

Además hay que tener en cuenta que es necesario añadir una coma para tuplas de un solo elemento, para diferenciarlo de un elemento entre paréntesis.

```
>>> t = (1)
>>> type(t)
type “int”
>>> t = (1,)
>>> type(t)
type “tuple”
```

Para referirnos a elementos de una tupla, como en una lista, se usa el operador `[]`:

```
mi_var = t[0] # mi_var es 1
mi_var = t[0:2] # mi_var es (1, 2)
```

Podemos utilizar el operador `[]` debido a que las tuplas, al igual que las listas, forman parte de un tipo de objetos llamados secuencias.
Permitirme un pequeño inciso para indicaros que las cadenas de texto también son secuencias, por lo que no os extrañará que podamos hacer cosas como estas:

```
c = “hola mundo”
c[0] # h
c[5:] # mundo
c[::3] # hauo
```

Volviendo al tema de las tuplas, su diferencia con las listas estriba en que las tuplas no poseen estos mecanismos de modificación a través de funciones tan útiles de los que hablábamos al final de la anterior sección.

Además son inmutables, es decir, sus valores no se pueden modificar una vez creada; y tienen un tamaño fijo.

A cambio de estas limitaciones las tuplas son más “ligeras” que las listas, por lo que si el uso que le vamos a dar a una colección es muy básico, puedes utilizar tuplas en lugar de listas y ahorrar memoria.

Diccionarios

Los diccionarios, también llamados matrices asociativas, deben su nombre a que son colecciones que relacionan una clave y un valor. Por ejemplo, veamos un diccionario de películas y directores:

```
d = {“Love Actually” : “Richard Curtis”,
     “Kill Bill”: “Tarantino”,
     “Amélie”: “Jean-Pierre Jeunet”}
```

El primer valor se trata de la clave y el segundo del valor asociado a la clave. Como clave podemos utilizar cualquier valor inmutable: podríamos usar números, cadenas, booleanos, tuplas, … pero no listas o diccionarios, dado que son mutables. Esto es así porque los diccionarios se implementan como tablas hash, y a la hora de introducir un nuevo par clave-valor en el diccionario se calcula el hash de la clave para después poder encontrar la entrada correspondiente rápidamente. Si se modificara el objeto clave después de haber sido introducido en el diccionario, evidentemente, su hash también cambiaría y no podría ser encontrado.

La diferencia principal entre los diccionarios y las listas o las tuplas es que a los valores almacenados en un diccionario se les accede no por su índice, porque de hecho no tienen orden, sino por su clave, utilizando de nuevo el operador `[]`.

```
d[“Love Actually”] # devuelve “Richard Curtis”
```

Al igual que en listas y tuplas también se puede utilizar este operador para reasignar valores.

```
d[“Kill Bill”] = “Quentin Tarantino”
```

Sin embargo en este caso no se puede utilizar slicing, entre otras cosas porque los diccionarios no son secuencias, si no *mappings* (mapeados, asociaciones).

CONTROL DE FLUJO

En esta lección vamos a ver los condicionales y los bucles.

Sentencias condicionales

Si un programa no fuera más que una lista de órdenes a ejecutar de forma secuencial, una por una, no tendría mucha utilidad. Los condicionales nos permiten comprobar condiciones y hacer que nuestro programa se comporte de una forma u otra, que ejecute un fragmento de código u otro, dependiendo de esta condición.

Aquí es donde cobran su importancia el tipo booleano y los operadores lógicos y relacionales que aprendimos en el capítulo sobre los tipos básicos de Python.

if

La forma más simple de un estamento condicional es un `if` (del inglés si) seguido de la condición a evaluar, dos puntos (`:`) y en la siguiente línea e indentado, el código a ejecutar en caso de que se cumpla dicha condición.

```
fav = “mundogeek.net”
# si (if) fav es igual a “mundogeek.net”
if fav == “mundogeek.net”:
  print “Tienes buen gusto!”
  print “Gracias”
```

Como veis es bastante sencillo.

Eso si, aseguraros de que indentáis el código tal cual se ha hecho en el ejemplo, es decir, aseguraros de pulsar Tabulación antes de las dos órdenes `print`, dado que esta es la forma de Python de saber que vuestra intención es la de que los dos `print` se ejecuten sólo en el caso de que se cumpla la condición, y no la de que se imprima la primera cadena si se cumple la condición y la otra siempre, cosa que se expresaría así:

```
if fav == “mundogeek.net”:
  print “Tienes buen gusto!”
print “Gracias”
```

En otros lenguajes de programación los bloques de código se determinan encerrándolos entre llaves, y el indentarlos no se trata más que de una buena práctica para que sea más sencillo seguir el flujo del programa con un solo golpe de vista. Por ejemplo, el código anterior expresado en Java sería algo así:

```
String fav = “mundogeek.net”;
if (fav.equals(“mundogeek.net”)){
  System.out.println(“Tienes buen gusto!”);
  System.out.println(“Gracias”);
}
```

Sin embargo, como ya hemos comentado, en Python se trata de una obligación, y no de una elección. De esta forma se obliga a los programadores a indentar su código para que sea más sencillo de leer :)

if … else

Vamos a ver ahora un condicional algo más complicado. ¿Qué haríamos si quisiéramos que se ejecutaran unas ciertas órdenes en el caso de que la condición no se cumpliera? Sin duda podríamos añadir otro `if` que tuviera como condición la negación del primero:

```
if fav == "mundogeek.net":
  print "Tienes buen gusto!"
  print "Gracias"

if fav != "mundogeek.net":
  print "Vaya, que lástima"
```

pero el condicional tiene una segunda construcción mucho más útil:

```
if fav == "mundogeek.net":
  print "Tienes buen gusto!"
  print "Gracias"
else:
  print "Vaya, que lástima"
```

Vemos que la segunda condición se puede sustituir con un `else` (del inglés: si no, en caso contrario). Si leemos el código vemos que tiene bastante sentido: "si `fav` es igual a mundogeek.net, imprime esto y esto, si no, imprime esto otro".

if … elif … elif … else

Todavía queda una construcción más que ver, que es la que hace uso del `elif`.

```
if numero < 0:
  print "Negativo"
elif numero > 0:
  print "Positivo"
else:
  print "Cero"
```

`elif` es una contracción de *else if*, por lo tanto `elif numero > 0` puede leerse como "si no, si numero es mayor que 0". Es decir, primero se evalúa la condición del `if`. Si es cierta, se ejecuta su código y se continúa ejecutando el código posterior al condicional; si no se cumple, se evalúa la condición del `elif`. Si se cumple la condición del `elif` se ejecuta su código y se continua ejecutando el código posterior al condicional; si no se cumple y hay más de un `elif` se continúa con el siguiente en orden de aparición. Si no se cumple la condición del `if` ni de ninguno de los `elif`, se ejecuta el código del `else`.

A if C else B

También existe una construcción similar al operador `?` de otros lenguajes, que no es más que una forma compacta de expresar un `if else`. En esta construcción se evalúa el predicado `C` y se devuelve `A` si se cumple o `B` si no se cumple: `A if C else B`.

Veamos un ejemplo:

```
var = “par” if (num % 2 == 0) else “impar”
```

Y eso es todo. Si conocéis otros lenguajes de programación puede que esperarais que os hablara ahora del `switch`, pero en Python no existe esta construcción, que podría emularse con un simple diccionario, así que pasemos directamente a los bucles.

Bucles

Mientras que los condicionales nos permiten ejecutar distintos fragmentos de código dependiendo de ciertas condiciones, los bucles nos permiten ejecutar un mismo fragmento de código un cierto número de veces, mientras se cumpla una determinada condición.

while

El bucle `while` (mientras) ejecuta un fragmento de código mientras se cumpla una condición.

```
edad = 0
while edad < 18:
  edad = edad + 1
  print “Felicidades, tienes “ + str(edad)
```

La variable `edad` comienza valiendo 0. Como la condición de que `edad` es menor que 18 es cierta (0 es menor que 18), se entra en el bucle. Se aumenta `edad` en 1 y se imprime el mensaje informando de que el usuario ha cumplido un año. Recordad que el operador + para las cadenas funciona concatenando ambas cadenas. Es necesario utilizar la función `str` (de *string*, cadena) para crear una cadena a partir del número, dado que no podemos concatenar números y cadenas, pero ya comentaremos esto y mucho más en próximos capítulos.

Ahora se vuelve a evaluar la condición, y 1 sigue siendo menor que 18, por lo que se vuelve a ejecutar el código que aumenta la edad en un año e imprime la edad en la pantalla. El bucle continuará ejecutándose hasta que `edad` sea igual a 18, momento en el cual la condición dejará de cumplirse y el programa continuaría ejecutando las instrucciones siguientes al bucle.

Ahora imaginemos que se nos olvidara escribir la instrucción que aumenta la edad. En ese caso nunca se llegaría a la condición de que `edad` fuese igual o mayor que 18, siempre sería 0, y el bucle continuaría indefinidamente escribiendo en pantalla `Has cumplido 0`.

Esto es lo que se conoce como un bucle infinito.

Sin embargo hay situaciones en las que un bucle infinito es útil. Por ejemplo, veamos un pequeño programa que repite todo lo que el usuario diga hasta que escriba `adios`.

```
while True:
  entrada = raw_input(“> “)
  if entrada == “adios”:
    break
  else:
    print entrada
```

Para obtener lo que el usuario escriba en pantalla utilizamos la función `raw_input`.

No es necesario que sepáis qué es una función ni cómo funciona exactamente, simplemente aceptad por ahora que en cada iteración del bucle la variable `entrada` contendrá lo que el usuario escribió hasta pulsar Enter.

Comprobamos entonces si lo que escribió el usuario fue `adios`, en cuyo caso se ejecuta la orden `break` o si era cualquier otra cosa, en cuyo caso se imprime en pantalla lo que el usuario escribió.

La palabra clave `break` (romper) sale del bucle en el que estamos.

Este bucle se podría haber escrito también, no obstante, de la siguiente forma:

```
salir = False
while not salir:
  entrada = raw_input()
  if entrada == “adios”:
    salir = True
  else:
    print entrada
```

pero nos ha servido para ver cómo funciona `break`.

Otra palabra clave que nos podemos encontrar dentro de los bucles es `continue` (continuar). Como habréis adivinado no hace otra cosa que pasar directamente a la siguiente iteración del bucle.

```
edad = 0
while edad < 18:
  edad = edad + 1
  if edad % 2 == 0:
    continue
  print “Felicidades, tienes “ + str(edad)
```

Como veis esta es una pequeña modificación de nuestro programa de felicitaciones. En esta ocasión hemos añadido un `if` que comprueba si la edad es par, en cuyo caso saltamos a la próxima iteración en lugar de imprimir el mensaje. Es decir, con esta modificación el programa sólo imprimiría felicitaciones cuando la edad fuera impar.

for … in

A los que hayáis tenido experiencia previa con según que lenguajes este bucle os va a sorprender gratamente. En Python `for` se utiliza como una forma genérica de iterar sobre una secuencia. Y como tal intenta facilitar su uso para este fin.

Este es el aspecto de un bucle `for` en Python:

```
secuencia = [“uno”, “dos”, “tres”]
for elemento in secuencia:
  print elemento
```

Como hemos dicho los `for` se utilizan en Python para recorrer secuencias, por lo que vamos a utilizar un tipo `secuencia`, como es la lista, para nuestro ejemplo.

Leamos la cabecera del bucle como si de lenguaje natural se tratara: “para cada

elemento en secuencia”. Y esto es exactamente lo que hace el bucle: para cada elemento que tengamos en la secuencia, ejecuta estas líneas de código.

Lo que hace la cabecera del bucle es obtener el siguiente elemento de la secuencia secuencia y almacenarlo en una variable de nombre elemento. Por esta razón en la primera iteración del bucle elemento valdrá “uno”, en la segunda “dos”, y en la tercera “tres”.

Fácil y sencillo.

En C o C++, por ejemplo, lo que habríamos hecho sería iterar sobre las posiciones, y no sobre los elementos:

```
int mi_array[] = {1, 2, 3, 4, 5};
int i;
for(i = 0; i < 5; i++) {
  printf(“%d\n”, mi_array[i]);
}
```

Es decir, tendríamos un bucle for que fuera aumentando una variable i en cada iteración, desde 0 al tamaño de la secuencia, y utilizaríamos esta variable a modo de índice para obtener cada elemento e imprimirlo.

Como veis el enfoque de Python es más natural e intuitivo.

Pero, ¿qué ocurre si quisiéramos utilizar el for como si estuviéramos en C o en Java, por ejemplo, para imprimir los números de 30 a 50? No os preocupéis, porque no necesitaríais crear una lista y añadir uno a uno los números del 30 al 50. Python proporciona una función llamada range (rango) que permite generar una lista que vaya desde el primer número que le indiquemos al segundo. Lo veremos después de ver al fin a qué se refiere ese término tan recurrente: las funciones.

FUNCIONES

Una función es un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor. A los fragmentos de código que tienen un nombre asociado y no devuelven valores se les suele llamar procedimientos. En Python no existen los procedimientos, ya que cuando el programador no especifica un valor de retorno la función devuelve el valor `None` (nada), equivalente al `null` de Java.

Además de ayudarnos a programar y depurar dividiendo el programa en partes las funciones también permiten reutilizar código.

En Python las funciones se declaran de la siguiente forma:

```
def mi_funcion(param1, param2):
  print param1
  print param2
```

Es decir, la palabra clave `def` seguida del nombre de la función y entre paréntesis los argumentos separados por comas. A continuación, en otra línea, indentado y después de los dos puntos tendríamos las líneas de código que conforman el código a ejecutar por la función.

También podemos encontrarnos con una cadena de texto como primera línea del cuerpo de la función. Estas cadenas se conocen con el nombre de *docstring* (cadena de documentación) y sirven, como su nombre indica, a modo de documentación de la función.

```
def mi_funcion(param1, param2):
  “““Esta funcion imprime los dos valores pasados
  como parametros”””
  print param1
  print param2
```

Esto es lo que imprime el operador `?` de iPython o la función `help` del lenguaje para proporcionar una ayuda sobre el uso y utilidad de las funciones. Todos los objetos pueden tener docstrings, no solo las funciones, como veremos más adelante.

Volviendo a la declaración de funciones, es importante aclarar que al declarar la función lo único que hacemos es asociar un nombre al fragmento de código que conforma la función, de forma que podamos ejecutar dicho código más tarde referenciándolo por su nombre. Es decir, a la hora de escribir estas líneas no se ejecuta la función. Para llamar a la función (ejecutar su código) se escribiría:

```
mi_funcion(“hola”, 2)
```

Es decir, el nombre de la función a la que queremos llamar seguido de los valores que queramos pasar como parámetros entre paréntesis. La asociación de los parámetros y los valores pasados a la función se hace normalmente de izquierda a derecha: como a `param1` le hemos dado un valor `“hola”` y `param2` vale 2, `mi_funcion` imprimiría `hola`

en una línea, y a continuación 2.

Sin embargo también es posible modificar el orden de los parámetros si indicamos el nombre del parámetro al que asociar el valor a la hora de llamar a la función:

```
mi_funcion(param2 = 2, param1 = “hola”)
```

El número de valores que se pasan como parámetro al llamar a la función tiene que coincidir con el número de parámetros que la función acepta según la declaración de la función. En caso contrario Python se quejará:

```
>>> mi_funcion(“hola”)
Traceback (most recent call last):
File “<stdin>”, line 1, in <module>
TypeError: mi_funcion() takes exactly 2 arguments (1 given)
```

También es posible, no obstante, definir funciones con un número variable de argumentos, o bien asignar valores por defecto a los parámetros para el caso de que no se indique ningún valor para ese parámetro al llamar a la función.

Los valores por defecto para los parámetros se definen situando un signo igual después del nombre del parámetro y a continuación el valor por defecto:

```
def imprimir(texto, veces = 1):
  print veces * texto
```

En el ejemplo anterior si no indicamos un valor para el segundo parámetro se imprimirá una sola vez la cadena que le pasamos como primer parámetro:

```
>>> imprimir(“hola”)
hola
```

si se le indica otro valor, será este el que se utilice:

```
>>> imprimir(“hola”, 2)
holahola
```

Para definir funciones con un número variable de argumentos colocamos un último parámetro para la función cuyo nombre debe precederse de un signo `*`:

```
def varios(param1, param2, *otros):
  for val in otros:
    print val

varios(1, 2)
varios(1, 2, 3)
varios(1, 2, 3, 4)
```

Esta sintaxis funciona creando una tupla (de nombre `otros` en el ejemplo) en la que se almacenan los valores de todos los parámetros extra pasados como argumento. Para la primera llamada, `varios(1, 2)`, la tupla `otros` estaría vacía dado que no se han pasado más parámetros que los dos definidos por defecto, por lo tanto no se imprimiría nada. En la segunda llamada `otros` valdría `(3, )`, y en la tercera `(3, 4)`.

También se puede preceder el nombre del último parámetro con `**`, en cuyo caso en lugar de una tupla se utilizaría un diccionario. Las claves de este diccionario serían

los nombres de los parámetros indicados al llamar a la función y los valores del diccionario, los valores asociados a estos parámetros.

En el siguiente ejemplo se utiliza la función `items` de los diccionarios, que devuelve una lista con sus elementos, para imprimir los parámetros que contiene el diccionario.

```
def varios(param1, param2, **otros):
  for i in otros.items():
    print i

varios(1, 2, tercero = 3)
```

Los que conozcáis algún otro lenguaje de programación os estaréis preguntando si en Python al pasar una variable como argumento de una función estas se pasan por referencia o por valor. En el paso por referencia lo que se pasa como argumento es una referencia o puntero a la variable, es decir, la dirección de memoria en la que se encuentra el contenido de la variable, y no el contenido en si. En el paso por valor, por el contrario, lo que se pasa como argumento es el valor que contenía la variable.

La diferencia entre ambos estriba en que en el paso por valor los cambios que se hagan sobre el parámetro no se ven fuera de la función, dado que los argumentos de la función son variables locales a la función que contienen los valores indicados por las variables que se pasaron como argumento. Es decir, en realidad lo que se le pasa a la función son copias de los valores y no las variables en si.

Si quisiéramos modificar el valor de uno de los argumentos y que estos cambios se reflejaran fuera de la función tendríamos que pasar el parámetro por referencia.

En C los argumentos de las funciones se pasan por valor, aunque se puede simular el paso por referencia usando punteros. En Java también se usa paso por valor, aunque para las variables que son objetos lo que se hace es pasar por valor la referencia al objeto, por lo que en realidad parece paso por referencia.

En Python también se utiliza el paso por valor de referencias a objetos, como en Java, aunque en el caso de Python, a diferencia de Java, todo es un objeto (para ser exactos lo que ocurre en realidad es que al objeto se le asigna otra etiqueta o nombre en el espacio de nombres local de la función).

Sin embargo no todos los cambios que hagamos a los parámetros dentro de una función Python se reflejarán fuera de esta, ya que hay que tener en cuenta que en Python existen objetos inmutables, como las tuplas, por lo que si intentáramos modificar una tupla pasada como parámetro lo que ocurriría en realidad es que se crearía una nueva instancia, por lo que los cambios no se verían fuera de la función.

Veamos un pequeño programa para demostrarlo. En este ejemplo se hace uso del método `append` de las listas. Un método no es más que una función que pertenece a un objeto, en este caso a una lista; y `append`, en concreto, sirve para añadir un elemento a una lista.

```
def f(x, y):
  x = x + 3
  y.append(23)
  print x, y

x = 22
y = [22]
f(x, y)
print x, y
```

El resultado de la ejecución de este programa sería

```
25 [22, 23]
22 [22, 23]
```

Como vemos la variable `x` no conserva los cambios una vez salimos de la función porque los enteros son inmutables en Python. Sin embargo la variable `y` si los conserva, porque las listas son mutables.

En resumen: los valores mutables se comportan como paso por referencia, y los inmutables como paso por valor.

Con esto terminamos todo lo relacionado con los parámetros de las funciones. Veamos por último cómo devolver valores, para lo que se utiliza la palabra clave `return`:

```
def sumar(x, y):
  return x + y

print sumar(3, 2)
```

Como vemos esta función tan sencilla no hace otra cosa que sumar los valores pasados como parámetro y devolver el resultado como valor de retorno.

También podríamos pasar varios valores que retornar a `return`.

```
def f(x, y):
  return x * 2, y * 2

a, b = f(1, 2)
```

Sin embargo esto no quiere decir que las funciones Python puedan devolver varios valores, lo que ocurre en realidad es que Python crea una tupla al vuelo cuyos elementos son los valores a retornar, y esta única variable es la que se devuelve.

ORIENTACIÓN A OBJETOS

En el capítulo de introducción ya comentábamos que Python es un lenguaje multiparadigma en el que se podía trabajar con programación estructurada, como veníamos haciendo hasta ahora, o con programación orientada a objetos o programación funcional.

La Programación Orientada a Objetos (POO u OOP según sus siglas en inglés) es un paradigma de programación en el que los conceptos del mundo real relevantes para nuestro problema se modelan a través de clases y objetos, y en el que nuestro programa consiste en una serie de interacciones entre estos objetos.

Clases y objetos

Para entender este paradigma primero tenemos que comprender qué es una clase y qué es un objeto. Un objeto es una entidad que agrupa un estado y una funcionalidad relacionadas. El estado del objeto se define a través de variables llamadas atributos, mientras que la funcionalidad se modela a través de funciones a las que se les conoce con el nombre de métodos del objeto.

Un ejemplo de objeto podría ser un coche, en el que tendríamos atributos como la marca, el número de puertas o el tipo de carburante y métodos como arrancar y parar. O bien cualquier otra combinación de atributos y métodos según lo que fuera relevante para nuestro programa.

Una clase, por otro lado, no es más que una plantilla genérica a partir de la cual instanciar los objetos; plantilla que es la que define qué atributos y métodos tendrán los objetos de esa clase.

Volviendo a nuestro ejemplo: en el mundo real existe un conjunto de objetos a los que llamamos coches y que tienen un conjunto de atributos comunes y un comportamiento común, esto es a lo que llamamos clase. Sin embargo, mi coche no es igual que el coche de mi vecino, y aunque pertenecen a la misma clase de objetos, son objetos distintos.

En Python las clases se definen mediante la palabra clave `class` seguida del nombre de la clase, dos puntos (:) y a continuación, indentado, el cuerpo de la clase. Como en el caso de las funciones, si la primera línea del cuerpo se trata de una cadena de texto, esta será la cadena de documentación de la clase o docstring.

```
class Coche:
  """Abstracción de los objetos coche."""
  def __init__(self, gasolina):
    self.gasolina = gasolina
    print "Tenemos", gasolina, "litros"

  def arrancar(self):
    if self.gasolina > 0:
      print "Arranca"
    else:
      print "No arranca"

  def conducir(self):
    if self.gasolina > 0:
      self.gasolina -= 1
      print "Quedan", self.gasolina, "litros"
    else:
      print "No se mueve"
```

Lo primero que llama la atención en el ejemplo anterior es el nombre tan curioso que tiene el método `__init__`. Este nombre es una convención y no un capricho. El método `__init__`, con una doble barra baja al principio y final del nombre, se ejecuta justo después de crear un nuevo objeto a partir de la clase, proceso que se conoce con

el nombre de instanciación. El método `__init__` sirve, como sugiere su nombre, para realizar cualquier proceso de inicialización que sea necesario.

Como vemos el primer parámetro de `__init__` y del resto de métodos de la clase es siempre `self`. Esta es una idea inspirada en Modula-3 y sirve para referirse al objeto actual. Este mecanismo es necesario para poder acceder a los atributos y métodos del objeto diferenciando, por ejemplo, una variable local `mi_var` de un atributo del objeto `self.mi_var`.

Si volvemos al método `__init__` de nuestra clase Coche veremos cómo se utiliza `self` para asignar al atributo gasolina del objeto (`self.gasolina`) el valor que el programador especificó para el parámetro `gasolina`. El parámetro `gasolina` se destruye al final de la función, mientras que el atributo `gasolina` se conserva (y puede ser accedido) mientras el objeto viva.

Para crear un objeto se escribiría el nombre de la clase seguido de cualquier parámetro que sea necesario entre paréntesis. Estos parámetros son los que se pasarán al método `__init__`, que como decíamos es el método que se llama al instanciar la clase.

```
mi_coche = Coche(3)
```

Os preguntareis entonces cómo es posible que a la hora de crear nuestro primer objeto pasemos un solo parámetro a `__init__`, el número 3, cuando la definición de la función indica claramente que precisa de dos parámetros (`self` y `gasolina`). Esto es así porque Python pasa el primer argumento (la referencia al objeto que se crea) automáticamente.

Ahora que ya hemos creado nuestro objeto, podemos acceder a sus atributos y métodos mediante la sintaxis `objeto.atributo` y `objeto.metodo()`:

```
>>> print mi_coche.gasolina
3
>>> mi_coche.arrancar()
Arranca
>>> mi_coche.conducir()
Quedan 2 litros
>>> mi_coche.conducir()
Quedan 1 litros
>>> mi_coche.conducir()
Quedan 0 litros
>>> mi_coche.conducir()
No se mueve
>>> mi_coche.arrancar()
No arranca
>>> print mi_coche.gasolina
0
```

Como último apunte recordar que en Python, como ya se comentó en repetidas ocasiones anteriormente, todo son objetos. Las cadenas, por ejemplo, tienen métodos como `upper()`, que devuelve el texto en mayúsculas o `count(sub)`, que devuelve el

número de veces que se encontró la cadena `sub` en el texto.

Herencia

Hay tres conceptos que son básicos para cualquier lenguaje de programación orientado a objetos: el encapsulamiento, la herencia y el polimorfismo.

En un lenguaje orientado a objetos cuando hacemos que una clase (subclase) herede de otra clase (superclase) estamos haciendo que la subclase contenga todos los atributos y métodos que tenía la superclase. No obstante al acto de heredar de una clase también se le llama a menudo “extender una clase”.

Supongamos que queremos modelar los instrumentos musicales de una banda, tendremos entonces una clase `Guitarra`, una clase `Bateria`, una clase `Bajo`, etc. Cada una de estas clases tendrá una serie de atributos y métodos, pero ocurre que, por el mero hecho de ser instrumentos musicales, estas clases compartirán muchos de sus atributos y métodos; un ejemplo sería el método `tocar()`.

Es más sencillo crear un tipo de objeto `Instrumento` con los atributos y métodos comunes e indicar al programa que `Guitarra`, `Bateria` y `Bajo` son tipos de instrumentos, haciendo que hereden de `Instrumento`.

Para indicar que una clase hereda de otra se coloca el nombre de la clase de la que se hereda entre paréntesis después del nombre de la clase:

```
class Instrumento:
  def __init__(self, precio):
    self.precio = precio

  def tocar(self):
    print “Estamos tocando musica”

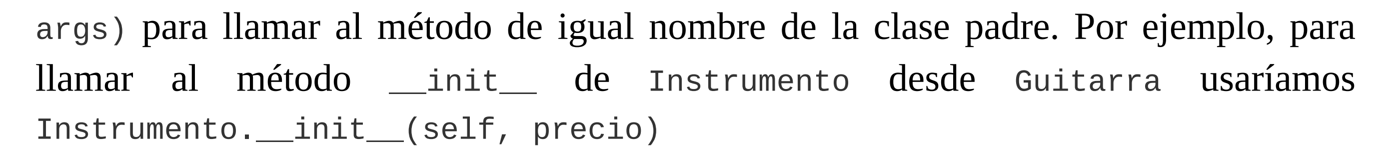
  def romper(self):
    print “Eso lo pagas tu”
    print “Son”, self.precio, “$$$”

  class Bateria(Instrumento):
    pass

  class Guitarra(Instrumento):
    pass
```

Como `Bateria` y `Guitarra` heredan de `Instrumento`, ambos tienen un método `tocar()` y un método `romper()`, y se inicializan pasando un parámetro `precio`. Pero, ¿qué ocurriría si quisiéramos especificar un nuevo parámetro `tipo_cuerda` a la hora de crear un objeto `Guitarra`? Bastaría con escribir un nuevo método `__init__` para la clase `Guitarra` que se ejecutaría en lugar del `__init__` de `Instrumento`. Esto es lo que se conoce como sobreescribir métodos.

Ahora bien, puede ocurrir en algunos casos que necesitemos sobreescribir un método de la clase padre, pero que en ese método queramos ejecutar el método de la clase padre porque nuestro nuevo método no necesite más que ejecutar un par de nuevas instrucciones extra. En ese caso usaríamos la sintaxis `SuperClase.metodo(self,`


args) para llamar al método de igual nombre de la clase padre. Por ejemplo, para llamar al método __init__ de Instrumento desde Guitarra usaríamos Instrumento.__init__(self, precio)

Observad que en este caso si es necesario especificar el parámetro self.

Herencia múltiple

En Python, a diferencia de otros lenguajes como Java o C#, se permite la herencia múltiple, es decir, una clase puede heredar de varias clases a la vez. Por ejemplo, podríamos tener una clase `Cocodrilo` que heredara de la clase `Terrestre`, con métodos como `caminar()` y atributos como `velocidad_caminar` y de la clase `Acuatico`, con métodos como `nadar()` y atributos como `velocidad_nadar`. Basta con enumerar las clases de las que se hereda separándolas por comas:

```
class Cocodrilo(Terrestre, Acuatico):
  pass
```

En el caso de que alguna de las clases padre tuvieran métodos con el mismo nombre y número de parámetros las clases sobreescribirían la implementación de los métodos de las clases más a su derecha en la definición.

En el siguiente ejemplo, como `Terrestre` se encuentra más a la izquierda, sería la definición de desplazar de esta clase la que prevalecería, y por lo tanto si llamamos al método desplazar de un objeto de tipo `Cocodrilo` lo que se imprimiría sería “El animal anda”.

```
class Terrestre:
  def desplazar(self):
    print “El animal anda”

class Acuatico:
  def desplazar(self):
    print “El animal nada”

class Cocodrilo(Terrestre, Acuatico):
  pass

c = Cocodrilo()
c.desplazar()
```

Polimorfismo

La palabra polimorfismo, del griego *poly morphos* (varias formas), se refiere a la habilidad de objetos de distintas clases de responder al mismo mensaje. Esto se puede conseguir a través de la herencia: un objeto de una clase derivada es al mismo tiempo un objeto de la clase padre, de forma que allí donde se requiere un objeto de la clase padre también se puede utilizar uno de la clase hija.

Python, al ser de tipado dinámico, no impone restricciones a los tipos que se le pueden pasar a una función, por ejemplo, más allá de que el objeto se comporte como se espera: si se va a llamar a un método `f()` del objeto pasado como parámetro, por ejemplo, evidentemente el objeto tendrá que contar con ese método. Por ese motivo, a diferencia de lenguajes de tipado estático como Java o C++, el polimorfismo en Python no es de gran importancia.

En ocasiones también se utiliza el término polimorfismo para referirse a la sobrecarga de métodos, término que se define como la capacidad del lenguaje de determinar qué método ejecutar de entre varios métodos con igual nombre según el tipo o número de los parámetros que se le pasa. En Python no existe sobrecarga de métodos (el último método sobreescribiría la implementación de los anteriores), aunque se puede conseguir un comportamiento similar recurriendo a funciones con valores por defecto para los parámetros o a la sintaxis `*params` o `**params` explicada en el capítulo sobre las funciones en Python, o bien usando decoradores (mecanismo que veremos más adelante).

Encapsulación

La encapsulación se refiere a impedir el acceso a determinados métodos y atributos de los objetos estableciendo así qué puede utilizarse desde fuera de la clase.

Esto se consigue en otros lenguajes de programación como Java utilizando modificadores de acceso que definen si cualquiera puede acceder a esa función o variable (`public`) o si está restringido el acceso a la propia clase (`private`).

En Python no existen los modificadores de acceso, y lo que se suele hacer es que el acceso a una variable o función viene determinado por su nombre: si el nombre comienza con dos guiones bajos (y no termina también con dos guiones bajos) se trata de una variable o función privada, en caso contrario es pública. Los métodos cuyo nombre comienza y termina con dos guiones bajos son métodos especiales que Python llama automáticamente bajo ciertas circunstancias, como veremos al final del capítulo.

En el siguiente ejemplo sólo se imprimirá la cadena correspondiente al método `publico()`, mientras que al intentar llamar al método `__privado()` Python lanzará una excepción quejándose de que no existe (evidentemente existe, pero no lo podemos ver porque es privado).

```
class Ejemplo:
  def publico(self):
    print “Publico”

  def __privado(self):
    print “Privado”

ej = Ejemplo()
ej.publico()
ej.__privado()
```

Este mecanismo se basa en que los nombres que comienzan con un doble guión bajo se renombran para incluir el nombre de la clase (característica que se conoce con el nombre de *name mangling*). Esto implica que el método o atributo no es realmente privado, y podemos acceder a él mediante una pequeña trampa:

```
ej._Ejemplo__privado()
```

En ocasiones también puede suceder que queramos permitir el acceso a algún atributo de nuestro objeto, pero que este se produzca de forma controlada. Para esto podemos escribir métodos cuyo único cometido sea este, métodos que normalmente, por convención, tienen nombres como `getVariable` y `setVariable`; de ahí que se conozcan también con el nombre de *getters* y *setters*.

```
class Fecha():
  def __init__(self):
    self.__dia = 1

  def getDia(self):
    return self.__dia
```

```
  def setDia(self, dia):
    if dia > 0 and dia < 31:
      self.__dia = dia
    else:
      print “Error”

mi_fecha = Fecha()
mi_fecha.setDia(33)
```

Esto se podría simplificar mediante propiedades, que abstraen al usuario del hecho de que se está utilizando métodos entre bambalinas para obtener y modificar los valores del atributo:

```
class Fecha(object):
  def __init__(self):
    self.__dia = 1

  def getDia(self):
    return self.__dia

  def setDia(self, dia):
    if dia > 0 and dia < 31:
      self.__dia = dia
    else:
      print “Error”

  dia = property(getDia, setDia)

mi_fecha = Fecha()
mi_fecha.dia = 33
```

Clases de "nuevo-estilo"

En el ejemplo anterior os habrá llamado la atención el hecho de que la clase `Fecha` derive de `object`. La razón de esto es que para poder usar propiedades la clase tiene que ser de "nuevo-estilo", clases enriquecidas introducidas en Python 2.2 que serán el estándar en Python 3.0 pero que aún conviven con las clases "clásicas" por razones de retrocompatibilidad. Además de las propiedades las clases de nuevo estilo añaden otras funcionalidades como descriptores o métodos estáticos.

Para que una clase sea de nuevo estilo es necesario, por ahora, que extienda una clase de nuevo-estilo. En el caso de que no sea necesario heredar el comportamiento o el estado de ninguna clase, como en nuestro ejemplo anterior, se puede heredar de `object`, que es un objeto vacío que sirve como base para todas las clases de nuevo estilo.

La diferencia principal entre las clases antiguas y las de nuevo estilo consiste en que a la hora de crear una nueva clase anteriormente no se definía realmente un nuevo tipo, sino que todos los objetos creados a partir de clases, fueran estas las clases que fueran, eran de tipo `instance`.

Métodos especiales

Ya vimos al principio del artículo el uso del método `__init__`. Existen otros métodos con significados especiales, cuyos nombres siempre comienzan y terminan con dos guiones bajos. A continuación se listan algunos especialmente útiles.

`__init__(self, args)`
Método llamado después de crear el objeto para realizar tareas de inicialización.

`__new__(cls, args)`
Método exclusivo de las clases de nuevo estilo que se ejecuta antes que `__init__` y que se encarga de construir y devolver el objeto en sí. Es equivalente a los constructores de C++ o Java. Se trata de un método estático, es decir, que existe con independencia de las instancias de la clase: es un método de clase, no de objeto, y por lo tanto el primer parámetro no es `self`, sino la propia clase: `cls`.

`__del__(self)`
Método llamado cuando el objeto va a ser borrado. También llamado destructor, se utiliza para realizar tareas de limpieza.

`__str__(self)`
Método llamado para crear una cadena de texto que represente a nuestro objeto. Se utiliza cuando usamos `print` para mostrar nuestro objeto o cuando usamos la función `str(obj)` para crear una cadena a partir de nuestro objeto.

`__cmp__(self, otro)`
Método llamado cuando se utilizan los operadores de comparación para comprobar si nuestro objeto es menor, mayor o igual al objeto pasado como parámetro. Debe devolver un número negativo si nuestro objeto es menor, cero si son iguales, y un número positivo si nuestro objeto es mayor. Si este método no está definido y se intenta comparar el objeto mediante los operadores `<`, `<=`, `>` o `>=` se lanzará una excepción. Si se utilizan los operadores `==` o `!=` para comprobar si dos objetos son iguales, se comprueba si son el mismo objeto (si tienen el mismo id).

`__len__(self)`
Método llamado para comprobar la longitud del objeto. Se utiliza, por ejemplo, cuando se llama a la función `len(obj)` sobre nuestro objeto. Como es de suponer, el método debe devolver la longitud del objeto.

Existen bastantes más métodos especiales, que permite entre otras cosas utilizar el mecanismo de *slicing* sobre nuestro objeto, utilizar los operadores aritméticos o usar la sintaxis de diccionarios, pero un estudio exhaustivo de todos los métodos queda fuera del propósito del capítulo.

REVISITANDO OBJETOS

En los capítulos dedicados a los tipos simples y las colecciones veíamos por primera vez algunos de los objetos del lenguaje Python: números, booleanos, cadenas de texto, diccionarios, listas y tuplas.

Ahora que sabemos qué son las clases, los objetos, las funciones, y los métodos es el momento de revisitar estos objetos para descubrir su verdadero potencial.

Veremos a continuación algunos métodos útiles de estos objetos. Evidentemente, no es necesario memorizarlos, pero sí, al menos, recordar que existen para cuando sean necesarios.

Diccionarios

`D.get(k[, d])`
Busca el valor de la clave `k` en el diccionario. Es equivalente a utilizar `D[k]` pero al utilizar este método podemos indicar un valor a devolver por defecto si no se encuentra la clave, mientras que con la sintaxis `D[k]`, de no existir la clave se lanzaría una excepción.

`D.has_key(k)`
Comprueba si el diccionario tiene la clave `k`. Es equivalente a la sintaxis `k in D`.

`D.items()`
Devuelve una lista de tuplas con pares clave-valor.

`D.keys()`
Devuelve una lista de las claves del diccionario.

`D.pop(k[, d])`
Borra la clave `k` del diccionario y devuelve su valor. Si no se encuentra dicha clave se devuelve `d` si se especificó el parámetro o bien se lanza una excepción.

`D.values()`
Devuelve una lista de los valores del diccionario.

Cadenas

`S.count(sub[, start[, end]])`
Devuelve el número de veces que se encuentra `sub` en la cadena. Los parámetros opcionales `start` y `end` definen una subcadena en la que buscar.

`S.find(sub[, start[, end]])`
Devuelve la posición en la que se encontró por primera vez `sub` en la cadena o `-1` si no se encontró.

`S.join(sequence)`
Devuelve una cadena resultante de concatenar las cadenas de la secuencia `sequence` separadas por la cadena sobre la que se llama el método.

`S.partition(sep)`
Busca el separador `sep` en la cadena y devuelve una tupla con la subcadena hasta dicho separador, el separador en si, y la subcadena del separador hasta el final de la cadena. Si no se encuentra el separador, la tupla contendrá la cadena en si y dos cadenas vacías.

`S.replace(old, new[, count])`
Devuelve una cadena en la que se han reemplazado todas las ocurrencias de la cadena `old` por la cadena `new`. Si se especifica el parámetro `count`, este indica el número máximo de ocurrencias a reemplazar.

`S.split([sep [,maxsplit]])`
Devuelve una lista conteniendo las subcadenas en las que se divide nuestra cadena al dividirlas por el delimitador `sep`. En el caso de que no se especifique `sep`, se usan espacios. Si se especifica `maxsplit`, este indica el número máximo de particiones a realizar.

Listas

`L.append(object)`
Añade un objeto al final de la lista.

`L.count(value)`
Devuelve el número de veces que se encontró `value` en la lista.

`L.extend(iterable)`
Añade los elementos del iterable a la lista.

`L.index(value[, start[, stop]])`
Devuelve la posición en la que se encontró la primera ocurrencia de `value`. Si se especifican, `start` y `stop` definen las posiciones de inicio y fin de una sublista en la que buscar.

`L.insert(index, object)`
Inserta el objeto `object` en la posición `index`.

`L.pop([index])`
Devuelve el valor en la posición `index` y lo elimina de la lista. Si no se especifica la posición, se utiliza el último elemento de la lista.

`L.remove(value)`
Eliminar la primera ocurrencia de `value` en la lista.

`L.reverse()`
Invierte la lista. Esta función trabaja sobre la propia lista desde la que se invoca el método, no sobre una copia.

`L.sort(cmp=None, key=None, reverse=False)`
Ordena la lista. Si se especifica `cmp`, este debe ser una función que tome como parámetro dos valores `x` e `y` de la lista y devuelva `-1` si `x` es menor que `y`, `0` si son iguales y 1 si `x` es mayor que `y`.

El parámetro `reverse` es un booleano que indica si se debe ordenar la lista de forma inversa, lo que sería equivalente a llamar primero a `L.sort()` y después a `L.reverse()`.

Por último, si se especifica, el parámetro `key` debe ser una función que tome un elemento de la lista y devuelva una clave a utilizar a la hora de comparar, en lugar del elemento en si.

PROGRAMACIÓN FUNCIONAL

La programación funcional es un paradigma en el que la programación se basa casi en su totalidad en funciones, entendiendo el concepto de función según su definición matemática, y no como los simples subprogramas de los lenguajes imperativos que hemos visto hasta ahora.

En los lenguajes funcionales puros un programa consiste exclusivamente en la aplicación de distintas funciones a un valor de entrada para obtener un valor de salida.

Python, sin ser un lenguaje puramente funcional incluye varias características tomadas de los lenguajes funcionales como son las funciones de orden superior o las funciones lambda (funciones anónimas).

Funciones de orden superior

El concepto de funciones de orden superior se refiere al uso de funciones como si de un valor cualquiera se tratara, posibilitando el pasar funciones como parámetros de otras funciones o devolver funciones como valor de retorno.

Esto es posible porque, como hemos insistido ya en varias ocasiones, en Python todo son objetos. Y las funciones no son una excepción.

Veamos un pequeño ejemplo

```
def saludar(lang):
  def saludar_es():
    print “Hola”

  def saludar_en():
    print “Hi”

  def saludar_fr():
    print “Salut”

  lang_func = {“es”: saludar_es,
        “en”: saludar_en,
        “fr”: saludar_fr}
  return lang_func[lang]

f = saludar(“es”)
f()
```

Como podemos observar lo primero que hacemos en nuestro pequeño programa es llamar a la función `saludar` con un parámetro `“es”`. En la función `saludar` se definen varias funciones: `saludar_es`, `saludar_en` y `saludar_fr` y a continuación se crea un diccionario que tiene como claves cadenas de texto que identifican a cada lenguaje, y como valores las funciones. El valor de retorno de la función es una de estas funciones. La función a devolver viene determinada por el valor del parámetro `lang` que se pasó como argumento de `saludar`.

Como el valor de retorno de `saludar` es una función, como hemos visto, esto quiere decir que `f` es una variable que contiene una función. Podemos entonces llamar a la función a la que se refiere `f` de la forma en que llamaríamos a cualquier otra función, añadiendo unos paréntesis y, de forma opcional, una serie de parámetros entre los paréntesis.

Esto se podría acortar, ya que no es necesario almacenar la función que nos pasan como valor de retorno en una variable para poder llamarla:

```
>>> saludar(“en”)()
Hi
>>> saludar(“fr”)()
Salut
```

En este caso el primer par de paréntesis indica los parámetros de la función `saludar`, y el segundo par, los de la función devuelta por `saludar`.

Iteraciones de orden superior sobre listas

Una de las cosas más interesantes que podemos hacer con nuestras funciones de orden superior es pasarlas como argumentos de las funciones `map`, `filter` y `reduce`. Estas funciones nos permiten sustituir los bucles típicos de los lenguajes imperativos mediante construcciones equivalentes.

map(function, sequence[, sequence, ...])

La función `map` aplica una función a cada elemento de una secuencia y devuelve una lista con el resultado de aplicar la función a cada elemento. Si se pasan como parámetros n secuencias, la función tendrá que aceptar n argumentos. Si alguna de las secuencias es más pequeña que las demás, el valor que le llega a la función `function` para posiciones mayores que el tamaño de dicha secuencia será `None`.

A continuación podemos ver un ejemplo en el que se utiliza `map` para elevar al cuadrado todos los elementos de una lista:

```
def cuadrado(n):
  return n ** 2

l = [1, 2, 3]
l2 = map(cuadrado, l)
```

filter(function, sequence)

La función `filter` verifica que los elementos de una secuencia cumplan una determinada condición, devolviendo una secuencia con los elementos que cumplen esa condición. Es decir, para cada elemento de `sequence` se aplica la función `function`; si el resultado es `True` se añade a la lista y en caso contrario se descarta.

A continuación podemos ver un ejemplo en el que se utiliza `filter` para conservar solo los números que son pares.

```
def es_par(n):
  return (n % 2.0 == 0)

l = [1, 2, 3]
l2 = filter(es_par, l)
```

reduce(function, sequence[, initial])

La función `reduce` aplica una función a pares de elementos de una secuencia hasta dejarla en un solo valor.

A continuación podemos ver un ejemplo en el que se utiliza `reduce` para sumar todos los elementos de una lista.

```
def sumar(x, y):
  return x + y
```

```
l = [1, 2, 3]
l2 = reduce(sumar, l)
```

Funciones lambda

El operador `lambda` sirve para crear funciones anónimas en línea. Al ser funciones anónimas, es decir, sin nombre, estas no podrán ser referenciadas más tarde.

Las funciones lambda se construyen mediante el operador `lambda`, los parámetros de la función separados por comas (atención, SIN paréntesis), dos puntos (`:`) y el código de la función.

Esta construcción podrían haber sido de utilidad en los ejemplos anteriores para reducir código. El programa que utilizamos para explicar `filter`, por ejemplo, podría expresarse así:

```
l = [1, 2, 3]
l2 = filter(lambda n: n % 2.0 == 0, l)
```

Comparémoslo con la versión anterior:

```
def es_par(n):
  return (n % 2.0 == 0)

l = [1, 2, 3]
l2 = filter(es_par, l)
```

Las funciones lambda están restringidas por la sintaxis a una sola expresión.

Comprensión de listas

En Python 3 `map`, `filter` y `reduce` perderán importancia. Y aunque estas funciones se mantendrán, `reduce` pasará a formar parte del módulo `functools`, con lo que quedará fuera de las funciones disponibles por defecto, y `map` y `filter` se desaconsejarán en favor de las *list comprehensions* o comprensión de listas.

La comprensión de listas es una característica tomada del lenguaje de programación funcional Haskell que está presente en Python desde la versión 2.0 y consiste en una construcción que permite crear listas a partir de otras listas.

Cada una de estas construcciones consta de una expresión que determina cómo modificar el elemento de la lista original, seguida de una o varias clausulas `for` y opcionalmente una o varias clausulas `if`.

Veamos un ejemplo de cómo se podría utilizar la comprensión de listas para elevar al cuadrado todos los elementos de una lista, como hicimos en nuestro ejemplo de `map`.

```
l2 = [n ** 2 for n in l]
```

Esta expresión se leería como “para cada `n` en `l` haz `n ** 2`”. Como vemos tenemos primero la expresión que modifica los valores de la lista original (`n ** 2`), después el `for`, el nombre que vamos a utilizar para referirnos al elemento actual de la lista original, el `in`, y la lista sobre la que se itera.

El ejemplo que utilizamos para la función `filter` (conservar solo los números que son pares) se podría expresar con comprensión de listas así:

```
l2 = [n for n in l if n % 2.0 == 0]
```

Veamos por último un ejemplo de compresión de listas con varias clausulas `for`:

```
l = [0, 1, 2, 3]
m = [“a”, “b”]
n = [s * v for s in m
     for v in l
     if v > 0]
```

Esta construcción sería equivalente a una serie de `for`-`in` anidados:

```
l = [0, 1, 2, 3]
m = [“a”, “b”]
n = []

for s in m:
  for v in l:
    if v > 0:
      n.append(s* v)
```

Generadores

Las expresiones generadoras funcionan de forma muy similar a la comprensión de listas. De hecho su sintaxis es exactamente igual, a excepción de que se utilizan paréntesis en lugar de corchetes:

```
l2 = (n ** 2 for n in l)
```

Sin embargo las expresiones generadoras se diferencian de la comprensión de listas en que no se devuelve una lista, sino un generador.

```
>>> l2 = [n ** 2 for n in l]
>>> l2
[0, 1, 4, 9]
>>> l2 = (n ** 2 for n in l)
>>> l2
<generator object at 0×00E33210>
```

Un generador es una clase especial de función que genera valores sobre los que iterar. Para devolver el siguiente valor sobre el que iterar se utiliza la palabra clave `yield` en lugar de `return`. Veamos por ejemplo un generador que devuelva números de `n` a `m` con un salto `s`.

```
def mi_generador(n, m, s):
  while(n <= m):
    yield n
    n += s

>>> x = mi_generador(0, 5, 1)
>>> x
<generator object at 0×00E25710>
```

El generador se puede utilizar en cualquier lugar donde se necesite un objeto iterable. Por ejemplo en un `for-in`:

```
for n in mi_generador(0, 5, 1):
  print n
```

Como no estamos creando una lista completa en memoria, sino generando un solo valor cada vez que se necesita, en situaciones en las que no sea necesario tener la lista completa el uso de generadores puede suponer una gran diferencia de memoria. En todo caso siempre es posible crear una lista a partir de un generador mediante la función `list`:

```
lista = list(mi_generador)
```

Decoradores

Un decorador no es es mas que una función que recibe una función como parámetro y devuelve otra función como resultado. Por ejemplo podríamos querer añadir la funcionalidad de que se imprimiera el nombre de la función llamada por motivos de depuración:

```
def mi_decorador(funcion):
  def nueva(*args):
    print “Llamada a la funcion”, funcion.__name__
    retorno = funcion(*args)
    return retorno
  return nueva
```

Como vemos el código de la función `mi_decorador` no hace más que crear una nueva función y devolverla. Esta nueva función imprime el nombre de la función a la que “decoramos”, ejecuta el código de dicha función, y devuelve su valor de retorno. Es decir, que si llamáramos a la nueva función que nos devuelve `mi_decorador`, el resultado sería el mismo que el de llamar directamente a la función que le pasamos como parámetro, exceptuando el que se imprimiría además el nombre de la función.

Supongamos como ejemplo una función `imp` que no hace otra cosa que mostrar en pantalla la cadena pasada como parámetro.

```
>>> imp(“hola”)
hola
>>> mi_decorador(imp)(“hola”)
Llamada a la función imp
hola
```

La sintaxis para llamar a la función que nos devuelve mi_decorador no es muy clara, aunque si lo estudiamos detenidamente veremos que no tiene mayor complicación. Primero se llama a la función que decora con la función a decorar: `mi_decorador(imp)`; y una vez obtenida la función ya decorada se la puede llamar pasando el mismo parámetro que se pasó anteriormente: `mi_decorador(imp)(“hola”)`

Esto se podría expresar más claramente precediendo la definición de la función que queremos decorar con el signo `@` seguido del nombre de la función decoradora:

```
@mi_decorador
def imp(s):
  print s
```

De esta forma cada vez que se llame a `imp` se estará llamando realmente a la versión decorada. Python incorpora esta sintaxis desde la versión 2.4 en adelante.

Si quisiéramos aplicar más de un decorador bastaría añadir una nueva línea con el nuevo decorador.

```
@otro_decorador
@mi_decorador
```

```
def imp(s):
  print s
```

Es importante advertir que los decoradores se ejecutarán de abajo a arriba. Es decir, en este ejemplo primero se ejecutaría `mi_decorador` y después `otro_decorador`.

Las excepciones son errores detectados por Python durante la ejecución del programa. Cuando el intérprete se encuentra con una situación excepcional, como el intentar dividir un número entre 0 o el intentar acceder a un archivo que no existe, este genera o lanza una excepción, informando al usuario de que existe algún problema.

Si la excepción no se captura el flujo de ejecución se interrumpe y se muestra la información asociada a la excepción en la consola de forma que el programador pueda solucionar el problema.

Veamos un pequeño programa que lanzaría una excepción al intentar dividir 1 entre 0.

```
def division(a, b):
  return a / b

def calcular():
  division(1, 0)

calcular()
```

Si lo ejecutamos obtendremos el siguiente mensaje de error:

```
$ python ejemplo.py
Traceback (most recent call last):
File "ejemplo.py", line 7, in
calcular()
File "ejemplo.py", line 5, in calcular
division(1, 0)
File "ejemplo.py", line 2, in division
a / b
ZeroDivisionError: integer division or modulo by zero
```

Lo primero que se muestra es el trazado de pila o *traceback*, que consiste en una lista con las llamadas que provocaron la excepción. Como vemos en el trazado de pila, el error estuvo causado por la llamada a `calcular()` de la línea 7, que a su vez llama a `division(1, 0)` en la línea 5 y en última instancia por la ejecución de la sentencia `a / b` de la línea 2 de `division`.

A continuación vemos el tipo de la excepción, `ZeroDivisionError`, junto a una descripción del error: "integer division or modulo by zero" (módulo o división entera entre cero).

En Python se utiliza una construcción `try`-`except` para capturar y tratar las excepciones. El bloque `try` (intentar) define el fragmento de código en el que creemos que podría producirse una excepción. El bloque `except` (excepción) permite indicar el tratamiento que se llevará a cabo de producirse dicha excepción. Muchas veces nuestro tratamiento de la excepción consistirá simplemente en imprimir un mensaje más amigable para el usuario, otras veces nos interesará registrar los errores

y de vez en cuando podremos establecer una estrategia de resolución del problema.

En el siguiente ejemplo intentamos crear un objeto `f` de tipo fichero. De no existir el archivo pasado como parámetro, se lanza una excepción de tipo `IOError`, que capturamos gracias a nuestro `try`-`except`.

```
try:
  f = file("archivo.txt")
except:
  print "El archivo no existe"
```

Python permite utilizar varios `except` para un solo bloque `try`, de forma que podamos dar un tratamiento distinto a la excepción dependiendo del tipo de excepción de la que se trate. Esto es una buena práctica, y es tan sencillo como indicar el nombre del tipo a continuación del `except`.

```
try:
  num = int("3a")
  print no_existe
except NameError:
  print "La variable no existe"
except ValueError:
  print "El valor no es un numero"
```

Cuando se lanza una excepción en el bloque `try`, se busca en cada una de las clausulas `except` un manejador adecuado para el tipo de error que se produjo. En caso de que no se encuentre, se propaga la excepción.

Además podemos hacer que un mismo `except` sirva para tratar más de una excepción usando una tupla para listar los tipos de error que queremos que trate el bloque:

```
try:
  num = int("3a")
  print no_existe
except (NameError, ValueError):
  print "Ocurrio un error"
```

La construcción `try`-`except` puede contar además con una clausula `else`, que define un fragmento de código a ejecutar sólo si no se ha producido ninguna excepción en el `try`.

```
try:
  num = 33
except:
  print "Hubo un error!"
else:
  print "Todo esta bien"
```

También existe una clausula `finally` que se ejecuta siempre, se produzca o no una excepción. Esta clausula se suele utilizar, entre otras cosas, para tareas de limpieza.

```
try:
  z = x / y
except ZeroDivisionError:
  print "Division por cero"
finally:
  print "Limpiando"
```

También es interesante comentar que como programadores podemos crear y lanzar nuestras propias excepciones. Basta crear una clase que herede de `Exception` o cualquiera de sus hijas y lanzarla con `raise`.

```
class MiError(Exception):
  def __init__(self, valor):
    self.valor = valor
  def __str__(self):
    return “Error “ + str(self.valor)

try:
  if resultado > 20:
    raise MiError(33)
except MiError, e:
  print e
```

Por último, a continuación se listan a modo de referencia las excepciones disponibles por defecto, así como la clase de la que deriva cada una de ellas entre paréntesis.

`BaseException`: Clase de la que heredan todas las excepciones.

`Exception(BaseException)`: Super clase de todas las excepciones que no sean de salida.

`GeneratorExit(Exception)`: Se pide que se salga de un generador.

`StandardError(Exception)`: Clase base para todas las excepciones que no tengan que ver con salir del intérprete.

`ArithmeticError(StandardError)`: Clase base para los errores aritméticos.

`FloatingPointError(ArithmeticError)`: Error en una operación de coma flotante.

`OverflowError(ArithmeticError)`: Resultado demasiado grande para poder representarse.

`ZeroDivisionError(ArithmeticError)`: Lanzada cuando el segundo argumento de una operación de división o módulo era 0.

`AssertionError(StandardError)`: Falló la condición de un estamento `assert`.

`AttributeError(StandardError)`: No se encontró el atributo.

`EOFError(StandardError)`: Se intentó leer más allá del final de fichero.

`EnvironmentError(StandardError)`: Clase padre de los errores relacionados con la entrada/salida.

`IOError(EnvironmentError)`: Error en una operación de entrada/salida.

`OSError(EnvironmentError)`: Error en una llamada a sistema.

`WindowsError(OSError)`: Error en una llamada a sistema en Windows.

`ImportError(StandardError)`: No se encuentra el módulo o el elemento del módulo que se quería importar.

`LookupError(StandardError)`: Clase padre de los errores de acceso.

`IndexError(LookupError)`: El índice de la secuencia está fuera del rango posible.

`KeyError(LookupError)`: La clave no existe.

`MemoryError(StandardError)`: No queda memoria suficiente.

`NameError(StandardError)`: No se encontró ningún elemento con ese nombre.

`UnboundLocalError(NameError)`: El nombre no está asociado a ninguna variable.

`ReferenceError(StandardError)`: El objeto no tiene ninguna referencia fuerte apuntando hacia él.

`RuntimeError(StandardError)`: Error en tiempo de ejecución no especificado.

`NotImplementedError(RuntimeError)`: Ese método o función no está implementado.

`SyntaxError(StandardError)`: Clase padre para los errores sintácticos.

`IndentationError(SyntaxError)`: Error en la indentación del archivo.

`TabError(IndentationError)`: Error debido a la mezcla de espacios y tabuladores.

`SystemError(StandardError)`: Error interno del intérprete.

`TypeError(StandardError)`: Tipo de argumento no apropiado.

`ValueError(StandardError)`: Valor del argumento no apropiado.

`UnicodeError(ValueError)`: Clase padre para los errores relacionados con unicode.

`UnicodeDecodeError(UnicodeError)`: Error de decodificación unicode.

`UnicodeEncodeError(UnicodeError)`: Error de codificación unicode.

`UnicodeTranslateError(UnicodeError)`: Error de traducción unicode.

`StopIteration(Exception)`: Se utiliza para indicar el final del iterador.

`Warning(Exception)`: Clase padre para los avisos.

`DeprecationWarning(Warning)`: Clase padre para avisos sobre características obsoletas.

`FutureWarning(Warning)`: Aviso. La semántica de la construcción cambiará en un futuro.

`ImportWarning(Warning)`: Aviso sobre posibles errores a la hora de importar.

`PendingDeprecationWarning(Warning)`: Aviso sobre características que se marcarán como obsoletas en un futuro próximo.

`RuntimeWarning(Warning)`: Aviso sobre comportamientos dudosos en tiempo de ejecución.

`SyntaxWarning(Warning)`: Aviso sobre sintaxis dudosa.

`UnicodeWarning(Warning)`: Aviso sobre problemas relacionados con Unicode, sobre todo con problemas de conversión.

`UserWarning(Warning)`: Clase padre para avisos creados por el programador.

`KeyboardInterrupt(BaseException)`: El programa fue interrumpido por el usuario.

`SystemExit(BaseException)`: Petición del intérprete para terminar la ejecución.

MÓDULOS Y PAQUETES

Módulos

Para facilitar el mantenimiento y la lectura los programas demasiado largos pueden dividirse en módulos, agrupando elementos relacionados. Los módulos son entidades que permiten una organización y división lógica de nuestro código. Los ficheros son su contrapartida física: cada archivo Python almacenado en disco equivale a un módulo.

Vamos a crear nuestro primer módulo entonces creando un pequeño archivo modulo.py con el siguiente contenido:

```
def mi_funcion():
  print “una funcion”

class MiClase:
  def __init__(self):
    print “una clase”

print “un modulo”
```

Si quisiéramos utilizar la funcionalidad definida en este módulo en nuestro programa tendríamos que importarlo. Para importar un módulo se utiliza la palabra clave `import` seguida del nombre del módulo, que consiste en el nombre del archivo menos la extensión. Como ejemplo, creemos un archivo programa.py en el mismo directorio en el que guardamos el archivo del módulo (esto es importante, porque si no se encuentra en el mismo directorio Python no podrá encontrarlo), con el siguiente contenido:

```
import modulo

modulo.mi_funcion()
```

El `import` no solo hace que tengamos disponible todo lo definido dentro del módulo, sino que también ejecuta el código del módulo. Por esta razón nuestro programa, además de imprimir el texto “una funcion” al llamar a `mi_funcion`, también imprimiría el texto “un modulo”, debido al `print` del módulo importado. No se imprimiría, no obstante, el texto “una clase”, ya que lo que se hizo en el módulo fue tan solo definir de la clase, no instanciarla.

La clausula `import` también permite importar varios módulos en la misma línea. En el siguiente ejemplo podemos ver cómo se importa con una sola clausula `import` los módulos de la distribución por defecto de Python `os`, que engloba funcionalidad relativa al sistema operativo; `sys`, con funcionalidad relacionada con el propio intérprete de Python y `time`, en el que se almacenan funciones para manipular fechas y horas.

```
import os, sys, time

print time.asctime()
```

Sin duda os habréis fijado en este y el anterior ejemplo en un detalle importante, y es que, como vemos, es necesario preceder el nombre de los objetos que importamos de un módulo con el nombre del módulo al que pertenecen, o lo que es lo mismo, el espacio de nombres en el que se encuentran. Esto permite que no sobreescribamos accidentalmente algún otro objeto que tuviera el mismo nombre al importar otro módulo.

Sin embargo es posible utilizar la construcción `from-import` para ahorrarnos el tener que indicar el nombre del módulo antes del objeto que nos interesa. De esta forma se importa el objeto o los objetos que indiquemos al espacio de nombres actual.

```
from time import asctime

print asctime()
```

Aunque se considera una mala práctica, también es posible importar todos los nombres del módulo al espacio de nombres actual usando el carácter `*`:

```
from time import *
```

Ahora bien, recordareis que a la hora de crear nuestro primer módulo insistí en que lo guardarais en el mismo directorio en el que se encontraba el programa que lo importaba. Entonces, ¿cómo podemos importar los módulos `os`, `sys` o `time` si no se encuentran los archivos `os.py`, `sys.py` y `time.py` en el mismo directorio?

A la hora de importar un módulo Python recorre todos los directorios indicados en la variable de entorno `PYTHONPATH` en busca de un archivo con el nombre adecuado. El valor de la variable `PYTHONPATH` se puede consultar desde Python mediante `sys.path`

```
>>> import sys
>>> sys.path
```

De esta forma para que nuestro módulo estuviera disponible para todos los programas del sistema bastaría con que lo copiáramos a uno de los directorios indicados en `PYTHONPATH`.

En el caso de que Python no encontrara ningún módulo con el nombre especificado, se lanzaría una excepción de tipo `ImportError`.

Por último es interesante comentar que en Python los módulos también son objetos; de tipo `module` en concreto. Por supuesto esto significa que pueden tener atributos y métodos. Uno de sus atributos, `__name__`, se utiliza a menudo para incluir código ejecutable en un módulo pero que este sólo se ejecute si se llama al módulo como programa, y no al importarlo. Para lograr esto basta saber que cuando se ejecuta el módulo directamente `__name__` tiene como valor `“__main__”`, mientras que cuando se importa, el valor de `__name__` es el nombre del módulo:

```
print “Se muestra siempre”

if __name__ == “__main__”:
```

```
print “Se muestra si no es importacion”
```

Otro atributo interesante es __doc__, que, como en el caso de funciones y clases, sirve a modo de documentación del objeto (docstring o cadena de documentación). Su valor es el de la primera línea del cuerpo del módulo, en el caso de que esta sea una cadena de texto; en caso contrario valdrá None.

Paquetes

Si los módulos sirven para organizar el código, los paquetes sirven para organizar los módulos. Los paquetes son tipos especiales de módulos (ambos son de tipo `module`) que permiten agrupar módulos relacionados. Mientras los módulos se corresponden a nivel físico con los archivos, los paquetes se representan mediante directorios.

En una aplicación cualquiera podríamos tener, por ejemplo, un paquete `iu` para la interfaz o un paquete `bbdd` para la persistencia a base de datos.

Para hacer que Python trate a un directorio como un paquete es necesario crear un archivo __init__.py en dicha carpeta. En este archivo se pueden definir elementos que pertenezcan a dicho paquete, como una constante `DRIVER` para el paquete `bbdd`, aunque habitualmente se tratará de un archivo vacío. Para hacer que un cierto módulo se encuentre dentro de un paquete, basta con copiar el archivo que define el módulo al directorio del paquete.

Como los módulos, para importar paquetes también se utiliza `import` y `from-import` y el carácter `.` para separar paquetes, subpaquetes y módulos.

```
import paq.subpaq.modulo

paq.subpaq.modulo.func()
```

A lo largo de los próximos capítulos veremos algunos módulos y paquetes de utilidad. Para encontrar algún módulo o paquete que cubra una cierta necesidad, puedes consultar la lista de PyPI (Python Package Index) en *http://pypi.python.org/*, que cuenta a la hora de escribir estas líneas, con más de 4000 paquetes distintos.

ENTRADA/SALIDA Y FICHEROS

Nuestros programas serían de muy poca utilidad si no fueran capaces de interaccionar con el usuario. En capítulos anteriores vimos, de pasada, el uso de la palabra clave `print` para mostrar mensajes en pantalla.

En esta lección, además de describir más detalladamente del uso de `print` para mostrar mensajes al usuario, aprenderemos a utilizar las funciones `input` y `raw_input` para pedir información, así como los argumentos de línea de comandos y, por último, la entrada/salida de ficheros.

Entrada estándar

La forma más sencilla de obtener información por parte del usuario es mediante la función `raw_input`. Esta función toma como parámetro una cadena a usar como prompt (es decir, como texto a mostrar al usuario pidiendo la entrada) y devuelve una cadena con los caracteres introducidos por el usuario hasta que pulsó la tecla Enter. Veamos un pequeño ejemplo:

```
nombre = raw_input(“Como te llamas? “)
print “Encantado, “ + nombre
```

Si necesitáramos un entero como entrada en lugar de una cadena, por ejemplo, podríamos utilizar la función `int` para convertir la cadena a entero, aunque sería conveniente tener en cuenta que puede lanzarse una excepción si lo que introduce el usuario no es un número.

```
try:
  edad = raw_input(“Cuantos anyos tienes? “)
  dias = int(edad) * 365
  print “Has vivido “ + str(dias) + “ dias”
except ValueError:
  print “Eso no es un numero”
```

La función `input` es un poco más complicada. Lo que hace esta función es utilizar `raw_input` para leer una cadena de la entrada estándar, y después pasa a evaluarla como si de código Python se tratara; por lo tanto `input` debería tratarse con sumo cuidado.

Parámetros de línea de comando

Además del uso de `input` y `raw_input` el programador Python cuenta con otros métodos para obtener datos del usuario. Uno de ellos es el uso de parámetros a la hora de llamar al programa en la línea de comandos. Por ejemplo:

```
python editor.py hola.txt
```

En este caso `hola.txt` sería el parámetro, al que se puede acceder a través de la lista `sys.argv`, aunque, como es de suponer, antes de poder utilizar dicha variable debemos importar el módulo `sys`. `sys.argv[0]` contiene siempre el nombre del programa tal como lo ha ejecutado el usuario, `sys.argv[1]`, si existe, sería el primer parámetro; `sys.argv[2]` el segundo, y así sucesivamente.

```
import sys

if(len(sys.argv) > 1):
  print “Abriendo “ + sys.argv[1]
else:
  print “Debes indicar el nombre del archivo”
```

Existen módulos, como `optparse`, que facilitan el trabajo con los argumentos de la línea de comandos, pero explicar su uso queda fuera del objetivo de este capítulo.

Salida estándar

La forma más sencilla de mostrar algo en la salida estándar es mediante el uso de la sentencia `print`, como hemos visto multitud de veces en ejemplos anteriores. En su forma más básica a la palabra clave `print` le sigue una cadena, que se mostrará en la salida estándar al ejecutarse el estamento.

```
>>> print “Hola mundo”
Hola mundo
```

Después de imprimir la cadena pasada como parámetro el puntero se sitúa en la siguiente línea de la pantalla, por lo que el `print` de Python funciona igual que el `println` de Java.

En algunas funciones equivalentes de otros lenguajes de programación es necesario añadir un carácter de nueva línea para indicar explícitamente que queremos pasar a la siguiente línea. Este es el caso de la función `printf` de C o la propia función `print` de Java.

Ya explicamos el uso de estos caracteres especiales durante la explicación del tipo cadena en el capítulo sobre los tipos básicos de Python. La siguiente sentencia, por ejemplo, imprimiría la palabra `“Hola”`, seguida de un renglón vacío (dos caracteres de nueva línea, `‘\n’`), y a continuación la palabra `“mundo”` indentada (un carácter tabulador, `‘\t’`).

```
print “Hola\n\n\tmundo”
```

Para que la siguiente impresión se realizara en la misma línea tendríamos que colocar una coma al final de la sentencia. Comparemos el resultado de este código:

```
>>> for i in range(3):
>>> ...print i,
0 1 2
```

Con el de este otro, en el que no utiliza una coma al final de la sentencia:

```
>>> for i in range(3):
>>> ...print i
0
1
2
```

Este mecanismo de colocar una coma al final de la sentencia funciona debido a que es el símbolo que se utiliza para separar cadenas que queramos imprimir en la misma línea.

```
>>> print “Hola”, “mundo”
Hola mundo
```

Esto se diferencia del uso del operador `+` para concatenar las cadenas en que al utilizar las comas `print` introduce automáticamente un espacio para separar cada una

de las cadenas. Este no es el caso al utilizar el operador `+`, ya que lo que le llega a `print` es un solo argumento: una cadena ya concatenada.

```
>>> print “Hola” + “mundo”
Holamundo
```

Además, al utilizar el operador `+` tendríamos que convertir antes cada argumento en una cadena de no serlo ya, ya que no es posible concatenar cadenas y otros tipos, mientras que al usar el primer método no es necesaria la conversión.

```
>>> print “Cuesta”, 3, “euros”
Cuesta 3 euros
>>> print “Cuesta” + 3 + “euros”
<type ‘exceptions.TypeError’>: cannot concatenate ‘str’ and
‘int’ objects
```

La sentencia `print`, o más bien las cadenas que imprime, permiten también utilizar técnicas avanzadas de formateo, de forma similar al `sprintf` de C. Veamos un ejemplo bastante simple:

```
print “Hola %s” % “mundo”
print “%s %s” % (“Hola”, “mundo”)
```

Lo que hace la primera línea es introducir los valores a la derecha del símbolo `%` (la cadena `“mundo”`) en las posiciones indicadas por los especificadores de conversión de la cadena a la izquierda del símbolo `%`, tras convertirlos al tipo adecuado.

En la segunda línea, vemos cómo se puede pasar más de un valor a sustituir, por medio de una tupla.

En este ejemplo sólo tenemos un especificador de conversión: `%s`.

Los especificadores más sencillos están formados por el símbolo `%` seguido de una letra que indica el tipo con el que formatear el valor. Por ejemplo:

Especificador	**Formato**
`%s`	Cadena
`%d`	Entero
`%o`	Octal
`%x`	Hexadecimal
`%f`	Real

Se puede introducir un número entre el `%` y el carácter que indica el tipo al que formatear, indicando el número mínimo de caracteres que queremos que ocupe la cadena. Si el tamaño de la cadena resultante es menor que este número, se añadirán espacios a la izquierda de la cadena. En el caso de que el número sea negativo, ocurrirá exactamente lo mismo, sólo que los espacios se añadirán a la derecha de la cadena.

```
>>> print “%10s mundo” % “Hola”
```

```
______Hola mundo
>>> print “%-10s mundo” % “Hola”
Hola_______mundo
```

En el caso de los reales es posible indicar la precisión a utilizar precediendo la `f` de un punto seguido del número de decimales que queremos mostrar:

```
>>> from math import pi
>>> print “%.4f” % pi
3.1416
```

La misma sintaxis se puede utilizar para indicar el número de caracteres de la cadena que queremos mostrar

```
>>> print “%.4s” % “hola mundo”
hola
```

Archivos

Los ficheros en Python son objetos de tipo `file` creados mediante la función `open` (abrir). Esta función toma como parámetros una cadena con la ruta al fichero a abrir, que puede ser relativa o absoluta; una cadena opcional indicando el modo de acceso (si no se especifica se accede en modo lectura) y, por último, un entero opcional para especificar un tamaño de buffer distinto del utilizado por defecto.

El modo de acceso puede ser cualquier combinación lógica de los siguientes modos:

- `'r'`: read, lectura. Abre el archivo en modo lectura. El archivo tiene que existir previamente, en caso contrario se lanzará una excepción de tipo `IOError`.
- `'w'`: write, escritura. Abre el archivo en modo escritura. Si el archivo no existe se crea. Si existe, sobreescribe el contenido.
- `'a'`: append, añadir. Abre el archivo en modo escritura. Se diferencia del modo `'w'` en que en este caso no se sobreescribe el contenido del archivo, sino que se comienza a escribir al final del archivo.
- `'b'`: binary, binario.
- `'+'`: permite lectura y escritura simultáneas.
- `'U'`: universal newline, saltos de línea universales. Permite trabajar con archivos que tengan un formato para los saltos de línea que no coincide con el de la plataforma actual (en Windows se utiliza el carácter `CR LF`, en Unix `LF` y en Mac OS `CR`).

```
f = open(“archivo.txt”, “w”)
```

Tras crear el objeto que representa nuestro archivo mediante la función open podremos realizar las operaciones de lectura/escritura pertinentes utilizando los métodos del objeto que veremos en las siguientes secciones.

Una vez terminemos de trabajar con el archivo debemos cerrarlo utilizando el método `close`.

Lectura de archivos

Para la lectura de archivos se utilizan los métodos `read`, `readline` y `readlines`.

El método `read` devuelve una cadena con el contenido del archivo o bien el contenido de los primeros n bytes, si se especifica el tamaño máximo a leer.

```
completo = f.read()

parte = f2.read(512)
```

El método `readline` sirve para leer las líneas del fichero una por una. Es decir, cada

vez que se llama a este método, se devuelve el contenido del archivo desde el puntero hasta que se encuentra un carácter de nueva línea, incluyendo este carácter.

```
while True:
  linea = f.readline()
  if not linea: break
  print linea
```

Por último, `readlines`, funciona leyendo todas las líneas del archivo y devolviendo una lista con las líneas leídas.

Escritura de archivos

Para la escritura de archivos se utilizan los métodos `write` y `writelines`. Mientras el primero funciona escribiendo en el archivo una cadena de texto que toma como parámetro, el segundo toma como parámetro una lista de cadenas de texto indicando las líneas que queremos escribir en el fichero.

Mover el puntero de lectura/escritura

Hay situaciones en las que nos puede interesar mover el puntero de lectura/escritura a una posición determinada del archivo. Por ejemplo si queremos empezar a escribir en una posición determinada y no al final o al principio del archivo.

Para esto se utiliza el método `seek` que toma como parámetro un número positivo o negativo a utilizar como desplazamiento. También es posible utilizar un segundo parámetro para indicar desde dónde queremos que se haga el desplazamiento: 0 indicará que el desplazamiento se refiere al principio del fichero (comportamiento por defecto), 1 se refiere a la posición actual, y 2, al final del fichero.

Para determinar la posición en la que se encuentra actualmente el puntero se utiliza el método `tell()`, que devuelve un entero indicando la distancia en bytes desde el principio del fichero.

EXPRESIONES REGULARES

Las expresiones regulares, también llamadas *regex* o *regexp*, consisten en patrones que describen conjuntos de cadenas de caracteres.

Algo parecido sería escribir en la línea de comandos de Windows

```
dir *.exe
```

‘`*.exe`’ sería una “expresión regular” que describiría todas las cadenas de caracteres que empiezan con cualquier cosa seguida de ‘.exe’, es decir, todos los archivos exe.

El trabajo con expresiones regulares en Python se realiza mediante el módulo `re`, que data de Python 1.5 y que proporciona una sintaxis para la creación de patrones similar a la de Perl. En Python 1.6 el módulo se reescribió para dotarlo de soporte de cadenas unicode y mejorar su rendimiento.

El módulo `re` contiene funciones para buscar patrones dentro de una cadena (`search`), comprobar si una cadena se ajusta a un determinado criterio descrito mediante un patrón (`match`), dividir la cadena usando las ocurrencias del patrón como puntos de ruptura (`split`) o para sustituir todas las ocurrencias del patrón por otra cadena (`sub`). Veremos estas funciones y alguna más en la próxima sección, pero por ahora, aprendamos algo más sobre la sintaxis de las expresiones regulares.

Patrones

La expresión regular más sencilla consiste en una cadena simple, que describe un conjunto compuesto tan solo por esa misma cadena. Por ejemplo, veamos cómo la cadena “`python`” coincide con la expresión regular “`python`” usando la función `match`:

```
import re

if re.match(“python”, “python”):
  print “cierto”
```

Si quisiéramos comprobar si la cadena es “`python`”, “`jython`”, “`cython`” o cualquier otra cosa que termine en “`ython`”, podríamos utilizar el carácter comodín, el punto ‘`.`’:

```
re.match(“.ython”, “python”)
re.match(“.ython”, “jython”)
```

La expresión regular “`.ython`” describiría a todas las cadenas que consistan en un carácter cualquiera, menos el de nueva línea, seguido de “`ython`”. Un carácter cualquiera y solo uno. No cero, ni dos, ni tres.

En el caso de que necesitáramos el carácter ‘`.`’ en la expresión regular, o cualquier otro de los caracteres especiales que veremos a continuación, tendríamos que escaparlo utilizando la barra invertida.

Para comprobar si la cadena consiste en 3 caracteres seguidos de un punto, por ejemplo, podríamos utilizar lo siguiente:

```
re.match(“...\.”, “abc.”)
```

Si necesitáramos una expresión que sólo resultara cierta para las cadenas “`python`”, “`jython`” y “`cython`” y ninguna otra, podríamos utilizar el carácter ‘`|`’ para expresar alternativa escribiendo los tres subpatrones completos:

```
re.match(“python|jython|cython”, “python”)
```

o bien tan solo la parte que pueda cambiar, encerrada entre paréntesis, formando lo que se conoce como un grupo. Los grupos tienen una gran importancia a la hora de trabajar con expresiones regulares y este no es su único uso, como veremos en la siguiente sección.

```
re.match(“(p|j|c)ython”, “python”)
```

Otra opción consistiría en encerrar los caracteres ‘`p`’, ‘`j`’ y ‘`c`’ entre corchetes para formar una clase de caracteres, indicando que en esa posición puede colocarse cualquiera de los caracteres de la clase.

```
re.match(“[pjc]ython”, “python”)
```

¿Y si quisiéramos comprobar si la cadena es “python0”, “python1”, “python2”, ... , “python9”? En lugar de tener que encerrar los 10 dígitos dentro de los corchetes podemos utilizar el guión, que sirve para indicar rangos. Por ejemplo `a-d` indicaría todas las letras minúsculas de la ‘`a`’ a la ‘`d`’; `0-9` serían todos los números de `0` a `9` inclusive.

```
re.match(“python[0-9]”, “python0”)
```

Si quisiéramos, por ejemplo, que el último carácter fuera o un dígito o una letra simplemente se escribirían dentro de los corchetes todos los criterios, uno detrás de otro.

```
re.match(“python[0-9a-zA-Z]”, “pythonp”)
```

Es necesario advertir que dentro de las clases de caracteres los caracteres especiales no necesitan ser escapados. Para comprobar si la cadena es “`python.`” o “`python,`”, entonces, escribiríamos:

```
re.match(“python[.,]”, “python.”)
```

y no

```
re.match(“python[\.,]”, “python.”)
```

ya que en este último caso estaríamos comprobando si la cadena es

```
“python.”, “python,” o “python\”.
```

Los conjuntos de caracteres también se pueden negar utilizando el símbolo ‘`^`’. La expresión “`python[^0-9a-z]`”, por ejemplo, indicaría que nos interesan las cadenas que comiencen por “`python`” y tengan como último carácter algo que no sea ni una letra minúscula ni un número.

```
re.match(“python[^0-9a-z]”, “python+”)
```

El uso de `[0-9]` para referirse a un dígito no es muy común, ya que, al ser la comprobación de que un carácter es un dígito algo muy utilizado, existe una secuencia especial equivalente: ‘`\d`’. Existen otras secuencias disponibles que listamos a continuación:

- “`\d`”: un dígito. Equivale a `[0-9]`
- “`\D`”: cualquier carácter que no sea un dígito. Equivale a `[^0-9]`
- “`\w`”: cualquier carácter alfanumérico. Equivale a `[a-zA-Z0-9_]`
- “`\W`”: cualquier carácter no alfanumérico. Equivale a `[^a-zA-Z0-9_]`
- “`\s`”: cualquier carácter en blanco. Equivale a `[ \t\n\r\f\v]`
- “`\S`”: cualquier carácter que no sea un espacio en blanco. Equivale a `[^ \t\n\r\f\v]`

Veamos ahora cómo representar repeticiones de caracteres, dado que no sería de

mucha utilidad tener que, por ejemplo, escribir una expresión regular con 30 caracteres '\d' para buscar números de 30 dígitos. Para este menester tenemos los caracteres especiales +, * y ?, además de las llaves {}.

El carácter + indica que lo que tenemos a la izquierda, sea un carácter como 'a', una clase como '[abc]' o un subpatrón como (abc), puede encontrarse una o mas veces. Por ejemplo la expresión regular "python+" describiría las cadenas "python", "pythonn" y "pythonnn", pero no "pytho", ya que debe haber al menos una n.

El carácter * es similar a +, pero en este caso lo que se sitúa a su izquierda puede encontrarse cero o mas veces.

El carácter ? indica opcionalidad, es decir, lo que tenemos a la izquierda puede o no aparecer (puede aparecer 0 o 1 veces).

Finalmente las llaves sirven para indicar el número de veces exacto que puede aparecer el carácter de la izquierda, o bien un rango de veces que puede aparecer. Por ejemplo {3} indicaría que tiene que aparecer exactamente 3 veces, {3,8} indicaría que tiene que aparecer de 3 a 8 veces, {,8} de 0 a 8 veces y {3,} tres veces o mas (las que sean).

Otro elemento interesante en las expresiones regulares, para terminar, es la especificación de las posiciones en que se tiene que encontrar la cadena, esa es la utilidad de ^ y $, que indican, respectivamente, que el elemento sobre el que actúan debe ir al principio de la cadena o al final de esta.

La cadena "http://mundogeek.net", por ejemplo, se ajustaría a la expresión regular "^http", mientras que la cadena "El protocolo es http" no lo haría, ya que el http no se encuentra al principio de la cadena.

Usando el módulo re

Ya hemos visto por encima cómo se utiliza la función `match` del módulo `re` para comprobar si una cadena se ajusta a un determinado patrón. El primer parámetro de la función es la expresión regular, el segundo, la cadena a comprobar y existe un tercer parámetro opcional que contiene distintos flags que se pueden utilizar para modificar el comportamiento de las expresiones regulares.

Algunos ejemplos de flags del módulo `re` son `re.IGNORECASE`, que hace que no se tenga en cuenta si las letras son mayúsculas o minúsculas o `re.VERBOSE`, que hace que se ignoren los espacios y los comentarios en la cadena que representa la expresión regular.

El valor de retorno de la función será `None` en caso de que la cadena no se ajuste al patrón o un objeto de tipo `MatchObject` en caso contrario. Este objeto `MatchObject` cuenta con métodos `start` y `end` que devuelven la posición en la que comienza y finaliza la subcadena reconocida y métodos `group` y `groups` que permiten acceder a los grupos que propiciaron el reconocimiento de la cadena.

Al llamar al método `group` sin parámetros se nos devuelve el grupo 0 de la cadena reconocida. El grupo 0 es la subcadena reconocida por la expresión regular al completo, aunque no existan paréntesis que delimiten el grupo.

```
>>> mo = re.match(“http://.+\net”, “http://mundogeek.net”)
>>> print mo.group()
http://mundogeek.net
```

Podríamos crear grupos utilizando los paréntesis, como aprendimos en la sección anterior, obteniendo así la parte de la cadena que nos interese.

```
>>> mo = re.match(“http://(.+)\net”, “http://mundogeek.net”)
>>> print mo.group(0)
http://mundogeek.net
>>> print mo.group(1)
mundogeek
```

El método `groups`, por su parte, devuelve una lista con todos los grupos, exceptuando el grupo 0, que se omite.

```
>>> mo = re.match(“http://(.+)\(.{3})”, “http://mundogeek.net”)
>>> print mo.groups()
(‘mundogeek’, ‘net’)
```

La función `search` del módulo `re` funciona de forma similar a `match`; contamos con los mismos parámetros y el mismo valor de retorno. La única diferencia es que al utilizar `match` la cadena debe ajustarse al patrón desde el primer carácter de la cadena, mientras que con `search` buscamos cualquier parte de la cadena que se ajuste al patrón. Por esta razón el método `start` de un objeto `MatchObject` obtenido mediante la función `match` siempre devolverá 0, mientras que en el caso de `search` esto no

tiene por qué ser así.

Otra función de búsqueda del módulo `re` es `findall`. Este toma los mismos parámetros que las dos funciones anteriores, pero devuelve una lista con las subcadenas que cumplieron el patrón.

Otra posibilidad, si no queremos todas las coincidencias, es utilizar `finditer`, que devuelve un iterador con el que consultar uno a uno los distintos `MatchObject`.

Las expresiones regulares no solo permiten realizar búsquedas o comprobaciones, sino que, como comentamos anteriormente, también tenemos funciones disponibles para dividir la cadena o realizar reemplazos.

La función `split` sin ir más lejos toma como parámetros un patrón, una cadena y un entero opcional indicando el número máximo de elementos en los que queremos dividir la cadena, y utiliza el patrón a modo de puntos de separación para la cadena, devolviendo una lista con las subcadenas.

La función `sub` toma como parámetros un patrón a sustituir, una cadena que usar como reemplazo cada vez que encontremos el patrón, la cadena sobre la que realizar las sustituciones, y un entero opcional indicando el número máximo de sustituciones que queremos realizar.

Al llamar a estos métodos lo que ocurre en realidad es que se crea un nuevo objeto de tipo `RegexObject` que representa la expresión regular, y se llama a métodos de este objeto que tienen los mismos nombres que las funciones del módulo.

Si vamos a utilizar un mismo patrón varias veces nos puede interesar crear un objeto de este tipo y llamar a sus métodos nosotros mismos; de esta forma evitamos que el intérprete tenga que crear un nuevo objeto cada vez que usemos el patrón y mejoraremos el rendimiento de la aplicación.

Para crear un objeto `RegexObject` se utiliza la función `compile` del módulo, al que se le pasa como parámetro la cadena que representa el patrón que queremos utilizar para nuestra expresión regular y, opcionalmente, una serie de flags de entre los que comentamos anteriormente.

SOCKETS

La comunicación entre distintas entidades en una red se basa en Python en el clásico concepto de *sockets*. Los sockets son un concepto abstracto con el que se designa al punto final de una conexión.

Los programas utilizan sockets para comunicarse con otros programas, que pueden estar situados en computadoras distintas.

Un socket queda definido por la dirección IP de la máquina, el puerto en el que escucha, y el protocolo que utiliza.

Los tipos y funciones necesarios para trabajar con sockets se encuentran en Python en el módulo `socket`, como no podría ser de otra forma.

Los sockets se clasifican en sockets de flujo (`socket.SOCK_STREAM`) o sockets de datagramas (`socket.SOCK_DGRAM`) dependiendo de si el servicio utiliza TCP, que es orientado a conexión y fiable, o UDP, respectivamente. En este capítulo sólo cubriremos los sockets de flujo, que cubren un 90% de las necesidades comunes.

Los sockets también se pueden clasificar según la familia. Tenemos sockets UNIX (`socket.AF_UNIX`) que se crearon antes de la concepción de las redes y se basan en ficheros, sockets `socket.AF_INET` que son los que nos interesan, sockets `socket.AF_INET6` para IPv6, etc.

Para crear un socket se utiliza el constructor `socket.socket()` que puede tomar como parámetros opcionales la familia, el tipo y el protocolo. Por defecto se utiliza la familia `AF_INET` y el tipo `SOCK_STREAM`.

Veremos durante el resto del capítulo cómo crear un par de programas cliente y servidor a modo de ejemplo.

Lo primero que tenemos que hacer es crear un objeto `socket` para el servidor

```
socket_s = socket.socket()
```

Tenemos ahora que indicar en qué puerto se va a mantener a la escucha nuestro servidor utilizando el método `bind`. Para sockets IP, como es nuestro caso, el argumento de `bind` es una tupla que contiene el host y el puerto. El host se puede dejar vacío, indicando al método que puede utilizar cualquier nombre que esté disponible.

```
socket_s.bind((“localhost”, 9999))
```

Por último utilizamos `listen` para hacer que el socket acepte conexiones entrantes y `accept` para comenzar a escuchar. El método `listen` requiere de un parámetro que indica el número de conexiones máximas que queremos aceptar; evidentemente, este

valor debe ser al menos 1.

El método `accept` se mantiene a la espera de conexiones entrantes, bloqueando la ejecución hasta que llega un mensaje.

Cuando llega un mensaje, `accept` desbloquea la ejecución, devolviendo un objeto `socket` que representa la conexión del cliente y una tupla que contiene el host y puerto de dicha conexión.

```
socket_s.listen(10)

socket_c, (host_c, puerto_c) = socket_s.accept()
```

Una vez que tenemos este objeto `socket` podemos comunicarnos con el cliente a través suyo, mediante los métodos `recv` y `send` (o `recvfrom` y `sendfrom` en UDP) que permiten recibir o enviar mensajes respectivamente. El método `send` toma como parámetros los datos a enviar, mientras que el método `recv` toma como parámetro el número máximo de bytes a aceptar.

```
recibido = socket_c.recv(1024)
print “Recibido: “, recibio
socket_c.send(recibido)
```

Una vez que hemos terminado de trabajar con el `socket`, lo cerramos con el método `close`.

Crear un cliente es aún más sencillo. Solo tenemos que crear el objeto `socket`, utilizar el método `connect` para conectarnos al servidor y utilizar los métodos `send` y `recv` que vimos anteriormente. El argumento de `connect` es una tupla con host y puerto, exactamente igual que `bind`.

```
socket_c = socket.socket()
socket_c.connect((“localhost”, 9999))
socket_c.send(“hola”)
```

Veamos por último un ejemplo completo. En este ejemplo el cliente manda al servidor cualquier mensaje que escriba el usuario y el servidor no hace más que repetir el mensaje recibido. La ejecución termina cuando el usuario escribe `quit`.

Este sería el código del script servidor:

```
import socket

s = socket.socket()
s.bind((“localhost”, 9999))
s.listen(1)

sc, addr = s.accept()

while True:
  recibido = sc.recv(1024)
  if recibido == “quit”:
    break
  print “Recibido:”, recibido
  sc.send(recibido)

print “adios”
```

```
sc.close()
s.close()
```

Y a continuación tenemos el del script cliente:

```
import socket

s = socket.socket()
s.connect((“localhost”, 9999))

while True:
  mensaje = raw_input(“> “)
  s.send(mensaje)
  mensaje == “quit”:
    break

print “adios”

s.close()
```

Existen dos módulos principales para leer datos de URLs en Python: `urllib` y `urllib2`. En esta lección aprenderemos a utilizar `urllib2` ya que es mucho más completo, aunque `urllib` tiene funcionalidades propias que no se pueden encontrar en `urllib2`, por lo que también lo tocaremos de pasada.

`urllib2` puede leer datos de una URL usando varios protocolos como HTTP, HTTPS, FTP, o Gopher.

Se utiliza una función `urlopen` para crear un objeto parecido a un fichero con el que leer de la URL. Este objeto cuenta con métodos como `read`, `readline`, `readlines` y `close`, los cuales funcionan exactamente igual que en los objetos `file`, aunque en realidad estamos trabajando con un wrapper que nos abstrae de un `socket` que se utiliza por debajo.

El método `read`, como recordareis, sirve para leer el "archivo" completo o el número de bytes especificado como parámetro, `readline` para leer una línea, y `readlines` para leer todas las líneas y devolver una lista con ellas.

También contamos con un par de métodos `geturl`, para obtener la URL de la que estamos leyendo (que puede ser útil para comprobar si ha habido una redirección) e `info` que nos devuelve un objeto con las cabeceras de respuesta del servidor (a las que también se puede acceder mediante el atributo `headers`).

```
import urllib2

try:
  f = urllib2.urlopen("http://www.python.org")
  print f.read()
  f.close()
except HTTPError, e:
  print "Ocurrió un error"
  print e.code
except URLError, e:
  print "Ocurrió un error"
  print e.reason
```

Al trabajar con `urllib2` nos podemos encontrar, como vemos, con errores de tipo `URLError`. Si trabajamos con HTTP podemos encontrarnos también con errores de la subclase de `URLError` `HTTPError`, que se lanzan cuando el servidor devuelve un código de error HTTP, como el error 404 cuando no se encuentra el recurso. También podríamos encontrarnos con errores lanzados por la librería que `urllib2` utiliza por debajo para las transferencias HTTP: `httplib`; o con excepciones lanzadas por el propio módulo `socket`.

La función `urlopen` cuenta con un parámetro opcional `data` con el que poder enviar información a direcciones HTTP (y solo HTTP) usando POST (los parámetros se

envían en la propia petición), por ejemplo para responder a un formulario. Este parámetro es una cadena codificada adecuadamente, siguiendo el formato utilizado en las URLs:

```
'password=contrase%A4a&usuario=manuel'
```

Lo más sencillo para codificar la cadena es utilizar el método `urlencode` de `urllib`, que acepta un diccionario o una lista de tuplas (clave, valor) y genera la cadena codificada correspondiente:

```
import urllib, urllib2

params = urllib.urlencode({"usuario": "manuel",
     "password": "contraseña"})
f = urllib2.urlopen("http://ejemplo.com/login", params)
```

Si lo único que queremos hacer es descargar el contenido de una URL a un archivo local, podemos utilizar la función `urlretrieve` de `urllib` en lugar de leer de un objeto creado con `urlopen` y escribir los datos leídos.

La función `urlretrieve` toma como parámetros la URL a descargar y, opcionalmente, un parámetro `filename` con la ruta local en la que guardar el archivo, un parámetro data similar al de `urlopen` y un parámetro `reporthook` con una función que utilizar para informar del progreso.

A excepción de las ocasiones en las que se utiliza el parámetro `data` las conexiones siempre se realizan utilizando GET (los parámetros se envían en la URL). Para enviar datos usando GET basta con concatenar la cadena resultante de `urlencode` con la URL a la que nos vamos a conectar mediante el símbolo `?`.

```
params = urllib.urlencode({"usuario": "manuel",
     "password": "contraseña"})

f = urllib2.urlopen("http://ejemplo.com/login" +
     "?" + params)
```

En `urllib` también se utiliza una función `urlopen` para crear nuestros pseudo-archivos, pero a diferencia de la versión de `urllib`, la función `urlopen` de `urllib2` también puede tomar como parámetro un objeto `Request`, en lugar de la URL y los datos a enviar.

La clase `Request` define objetos que encapsulan toda la información relativa a una petición. A través de este objeto podemos realizar peticiones más complejas, añadiendo nuestras propias cabeceras, como el User-Agent.

El constructor más sencillo para el objeto `Request` no toma más que una cadena indicando la URL a la que conectarse, por lo que utilizar este objeto como parámetro de `urlopen` sería equivalente a utilizar una cadena con la URL directamente.

Sin embargo el constructor de `Request` también tiene como parámetros opcionales

una cadena data para mandar datos por POST y un diccionario `headers` con las cabeceras (además de un par de campos `origin_req_host` y `unverifiable`, que quedan fuera del propósito del capítulo por ser de raro uso).

Veamos cómo añadir nuestras propias cabeceras utilizando como ejemplo la cabecera User-Agent. El User-Agent es una cabecera que sirve para identificar el navegador y sistema operativo que estamos utilizando para conectarnos a esa URL. Por defecto `urllib2` se identifica como "Python-urllib/2.5"; si quisiéramos identificarnos como un Linux corriendo Konqueror por ejemplo, usaríamos un código similar al siguiente:

```
ua = "Mozilla/5.0 (compatible; Konqueror/3.5.8; Linux)"
h = {"User-Agent": ua}
r = urllib2.Request("http://www.python.org", headers=h)
f = urllib2.urlopen(r)
print f.read()
```

Para personalizar la forma en que trabaja `urllib2` podemos instalar un grupo de manejadores (*handlers*) agrupados en un objeto de la clase `OpenerDirector` (*opener* o abridor), que será el que se utilice a partir de ese momento al llamar a `urlopen`.

Para construir un opener se utiliza la función `build_opener` a la que se le pasa los manejadores que formarán parte del opener. El opener se encargará de encadenar la ejecución de los distintos manejadores en el orden dado. También se puede usar el constructor de `OpenerDirector`, y añadir los manejadores usando su método `add_handler`.

Para instalar el opener una vez creado se utiliza la función `install_opener`, que toma como parámetro el opener a instalar. También se podría, si sólo queremos abrir la URL con ese opener una sola vez, utilizar el método `open` del opener.

`urllib2` cuenta con handlers que se encargan de manejar los esquemas disponibles (HTTP, HTTPS, FTP), manejar la autenticación, manejar las redirecciones, etc.

Para añadir autenticación tendríamos que instalar un opener que incluyera como manejador `HTTPBasicAuthHandler`, `ProxyBasicAuthHandler`, `HTTPDigestAuthHandler` y/o `ProxyDigestAuthHandler`.

Para utilizar autenticación HTTP básica, por ejemplo, usaríamos `HTTPBasicAuthHandler`:

```
aut_h = urllib2.HTTPBasicAuthHandler()
aut_h.add_password("realm", "host", "usuario", "password")

opener = urllib2.build_opener(aut_h)
urllib2.install_opener(opener)

f = urllib2.urlopen("http://www.python.org")
```

Si quisiéramos especificar un proxy en el código tendríamos que utilizar un opener que contuviera el manejador `ProxyHandler`. El manejador por defecto incluye una

instancia de `ProxyHandler` construido llamando al inicializador sin parámetros, con lo que se lee la lista de proxies a utilizar de la variable de entorno adecuada. Sin embargo también podemos construir un `ProxyHandler` pasando como parámetro al inicializador un diccionario cuyas claves son los protocolos y los valores, la URL del proxy a utilizar para dicho protocolo.

```
proxy_h = urllib2.ProxyHandler({“http” : “http://miproxy.net:123”})

opener = urllib2.build_opener(proxy_h)
urllib2.install_opener(opener)

f = urllib2.urlopen(“http://www.python.org”)
```

Para que se guarden las cookies que manda HTTP utilizamos el manejador `HTTPCookieProcessor`.

```
cookie_h = urllib2.HTTPCookieProcessor()

opener = urllib2.build_opener(cookie_h)
urllib2.install_opener(opener)

f = urllib2.urlopen(“http://www.python.org”)
```

Si queremos acceder a estas cookies o poder mandar nuestras propias cookies, podemos pasarle como parámetro al inicializador de `HTTPCookieProcessor` un objeto de tipo `CookieJar` del módulo `cookielib`.

Para leer las cookies mandadas basta crear un objeto iterable a partir del `CookieJar` (también podríamos buscar las cabeceras correspondientes, pero este sistema es más claro y sencillo):

```
import urllib2, cookielib

cookie_j = cookielib.CookieJar()

cookie_h = urllib2.HTTPCookieProcessor(cookie_j)

opener = urllib2.build_opener(cookie_h)
opener.open(“http://www.python.org”)

for num, cookie in enumerate(cookie_j):
  print num, cookie.name
  print cookie.value
  print
```

En el improbable caso de que necesitáramos añadir una cookie antes de realizar la conexión, en lugar de conectarnos para que el sitio la mande, podríamos utilizar el método `set_cookie` de `CookieJar`, al que le pasamos un objeto de tipo `Cookie`. El constructor de `Cookie`, no obstante, es bastante complicado.

¿Qué son los procesos y los threads?

Las computadoras serían mucho menos útiles si no pudiéramos hacer más de una cosa a la vez. Si no pudiéramos, por ejemplo, escuchar música en nuestro reproductor de audio favorito mientras leemos un tutorial de Python en Mundo Geek.

Pero, ¿cómo se conseguía esto en computadoras antiguas con un solo núcleo / una sola CPU? Lo que ocurría, y lo que ocurre ahora, es que en realidad no estamos ejecutando varios procesos a la vez (se llama proceso a un programa en ejecución), sino que los procesos se van turnando y, dada la velocidad a la que ejecutan las instrucciones, nosotros tenemos la impresión de que las tareas se ejecutan de forma paralela como si tuviéramos multitarea real.

Cada vez que un proceso distinto pasa a ejecutarse es necesario realizar lo que se llama un cambio de contexto, durante el cual se salva el estado del programa que se estaba ejecutando a memoria y se carga el estado del programa que va a entrar a ejecutarse.

En Python podemos crear nuevos procesos mediante la función `os.fork`, que ejecuta la llamada al sistema `fork`, o mediante otras funciones más avanzadas como `popen2.popen2`, de forma que nuestro programa pueda realizar varias tareas de forma paralela.

Sin embargo el cambio de contexto puede ser relativamente lento, y los recursos necesarios para mantener el estado demasiados, por lo que a menudo es mucho más eficaz utilizar lo que se conoce como *threads*, hilos de ejecución, o procesos ligeros.

Los threads son un concepto similar a los procesos: también se trata de código en ejecución. Sin embargo los threads se ejecutan dentro de un proceso, y los threads del proceso comparten recursos entre si, como la memoria, por ejemplo.

El sistema operativo necesita menos recursos para crear y gestionar los threads, y al compartir recursos, el cambio de contexto es más rápido. Además, dado que los threads comparten el mismo espacio de memoria global, es sencillo compartir información entre ellos: cualquier variable global que tengamos en nuestro programa es vista por todos los threads.

El GIL

La ejecución de los threads en Python está controlada por el *GIL* (Global Interpreter Lock) de forma que sólo un thread puede ejecutarse a la vez, independientemente del número de procesadores con el que cuente la máquina. Esto posibilita que el escribir extensiones en C para Python sea mucho más sencillo, pero tiene la desventaja de limitar mucho el rendimiento, por lo que a pesar de todo, en Python, en ocasiones nos puede interesar más utilizar procesos que threads, que no sufren de esta limitación.

Cada cierto número de instrucciones de bytecode la máquina virtual para la ejecución del thread y elige otro de entre los que estaban esperando.

Por defecto el cambio de thread se realiza cada 10 instrucciones de bytecode, aunque se puede modificar mediante la función `sys.setcheckinterval`. También se cambia de thread cuando el hilo se pone a dormir con `time.sleep` o cuando comienza una operación de entrada/salida, las cuales pueden tardar mucho en finalizar, y por lo tanto, de no realizar el cambio, tendríamos a la CPU demasiado tiempo sin trabajar esperando a que la operación de E/S terminara.

Para minimizar un poco el efecto del GIL en el rendimiento de nuestra aplicación es conveniente llamar al intérprete con el flag `-O`, lo que hará que se genere un bytecode optimizado con menos instrucciones, y, por lo tanto, menos cambios de contexto. También podemos plantearnos el utilizar procesos en lugar de threads, como ya comentamos, utilizando por ejemplo el módulo `processing`; escribir el código en el que el rendimiento sea crítico en una extensión en C o utilizar IronPython o Jython, que carecen de GIL.

Threads en Python

El trabajo con threads se lleva a cabo en Python mediante el módulo `thread`. Este módulo es opcional y dependiente de la plataforma, y puede ser necesario, aunque no es común, recompilar el intérprete para añadir el soporte de threads.

Además de `thread`, también contamos con el módulo `threading` que se apoya en el primero para proporcionarnos una API de más alto nivel, más completa, y orientada a objetos. El módulo `threading` se basa ligeramente en el modelo de threads de Java.

El módulo `threading` contiene una clase `Thread` que debemos extender para crear nuestros propios hilos de ejecución. El método `run` contendrá el código que queremos que ejecute el thread. Si queremos especificar nuestro propio constructor, este deberá llamar a `threading.Thread.__init__(self)` para inicializar el objeto correctamente.

```
import threading

class MiThread(threading.Thread):
  def __init__(self, num):
    threading.Thread.__init__(self)
    self.num = num

  def run(self):
    print “Soy el hilo”, self.num
```

Para que el thread comience a ejecutar su código basta con crear una instancia de la clase que acabamos de definir y llamar a su método `start`. El código del hilo principal y el del que acabamos de crear se ejecutarán de forma concurrente.

```
print “Soy el hilo principal”

for i in range(0, 10):
  t = MiThread(i)
  t.start()
  t.join()
```

El método `join` se utiliza para que el hilo que ejecuta la llamada se bloquee hasta que finalice el thread sobre el que se llama. En este caso se utiliza para que el hilo principal no termine su ejecución antes que los hijos, lo cual podría resultar en algunas plataformas en la terminación de los hijos antes de finalizar su ejecución. El método `join` puede tomar como parámetro un número en coma flotante indicando el número máximo de segundos a esperar.

Si se intenta llamar al método `start` para una instancia que ya se está ejecutando, obtendremos una excepción.

La forma recomendada de crear nuevos hilos de ejecución consiste en extender la clase `Thread`, como hemos visto, aunque también es posible crear una instancia de `Thread` directamente, e indicar como parámetros del constructor una clase ejecutable

(una clase con el método especial __call__) o una función a ejecutar, y los argumentos en una tupla (parámetro args) o un diccionario (parámetro kwargs).

```
import threading

def imprime(num):
  print “Soy el hilo”, num

print “Soy el hilo principal”

for i in range(0, 10):
  t = threading.Thread(target=imprime, args=(i, ))
  t.start()
```

Además de los parámetros target, args y kwargs también podemos pasar al constructor un parámetro de tipo cadena name con el nombre que queremos que tome el thread (el thread tendrá un nombre predeterminado aunque no lo especifiquemos); un parámetro de tipo booleano verbose para indicar al módulo que imprima mensajes sobre el estado de los threads para la depuración y un parámetro group, que por ahora no admite ningún valor pero que en el futuro se utilizará para crear grupos de threads y poder trabajar a nivel de grupos.

Para comprobar si un thread sigue ejecutándose, se puede utilizar el método isAlive. También podemos asignar un nombre al hilo y consultar su nombre con los métodos setName y getName, respectivamente.

Mediante la función threading.enumerate obtendremos una lista de los objetos Thread que se están ejecutando, incluyendo el hilo principal (podemos comparar el objeto Thread con la variable main_thread para comprobar si se trata del hilo principal) y con threading.activeCount podemos consultar el número de threads ejecutándose.

Los objetos Thread también cuentan con un método setDaemon que toma un valor booleano indicando si se trata de un demonio. La utilidad de esto es que si solo quedan threads de tipo demonio ejecutándose, la aplicación terminará automáticamente, terminando estos threads de forma segura.

Por último tenemos en el módulo threading una clase Timer que hereda de Thread y cuya utilidad es la de ejecutar el código de su método run después de un periodo de tiempo indicado como parámetro en su constructor. También incluye un método cancel mediante el que cancelar la ejecución antes de que termine el periodo de espera.

Sincronización

Uno de los mayores problemas a los que tenemos que enfrentarnos al utilizar threads es la necesidad de sincronizar el acceso a ciertos recursos por parte de los threads. Entre los mecanismos de sincronización que tenemos disponibles en el módulo `threading` se encuentran los locks, locks reentrantes, semáforos, condiciones y eventos.

Los locks, también llamados mutex (de *mutual exclusion*), cierres de exclusión mutua, cierres o candados, son objetos con dos estados posibles: adquirido o libre. Cuando un thread adquiere el candado, los demás threads que lleguen a ese punto posteriormente y pidan adquirirlo se bloquearán hasta que el thread que lo ha adquirido libere el candado, momento en el cual podrá entrar otro thread.

El candado se representa mediante la clase `Lock`. Para adquirir el candado se utiliza el método `acquire` del objeto, al que se le puede pasar un booleano para indicar si queremos esperar a que se libere (`True`) o no (`False`). Si indicamos que no queremos esperar, el método devolverá `True` o `False` dependiendo de si se adquirió o no el candado, respectivamente. Por defecto, si no se indica nada, el hilo se bloquea indefinidamente.

Para liberar el candado una vez hemos terminado de ejecutar el bloque de código en el que pudiera producirse un problema de concurrencia, se utiliza el método `release`.

```
lista = []

lock = threading.Lock()

def anyadir(obj):
  lock.acquire()
  lista.append(obj)
  lock.release()

def obtener():
  lock.acquire()
  obj = lista.pop()
  lock.release()
  return obj
```

La clase `RLock` funciona de forma similar a `Lock`, pero en este caso el candado puede ser adquirido por el mismo thread varias veces, y no quedará liberado hasta que el thread llame a `release` tantas veces como llamó a `acquire`. Como en `Lock`, y como en todas las primitivas de sincronización que veremos a continuación, es posible indicar a `acquire` si queremos que se bloquee o no.

Los semáforos son otra clase de candados. La clase correspondiente, `Semaphore`, también cuenta con métodos `acquire` y `release`, pero se diferencia de un `Lock` normal en que el constructor de `Semaphore` puede tomar como parámetro opcional un entero `value` indicando el número máximo de threads que pueden acceder a la vez a

la sección de código crítico. Si no se indica nada permite el acceso a un solo thread.

Cuando un thread llama a `acquire`, la variable que indica el número de threads que pueden adquirir el semáforo disminuye en 1, porque hemos permitido entrar en la sección de código crítico a un hilo más. Cuando un hilo llama a `release`, la variable aumenta en 1.

No es hasta que esta variable del semáforo es 0, que llamar a `acquire` producirá un bloqueo en el thread que realizó la petición, a la espera de que algún otro thread llame a `release` para liberar su plaza.

Es importante destacar que el valor inicial de la variable tal como lo pasamos en el constructor, no es un límite máximo, sino que múltiples llamadas a `release` pueden hacer que el valor de la variable sea mayor que su valor original. Si no es esto lo que queremos, podemos utilizar la clase `BoundedSemaphore` en cuyo caso, ahora si, se consideraría un error llamar a `release` demasiadas veces, y se lanzaría una excepción de tipo `ValueError` de superarse el valor inicial.

Podríamos utilizar los semáforos, por ejemplo, en un pequeño programa en el que múltiples threads descargaran datos de una URL, de forma que pudiéramos limitar el número de conexiones a realizar al sitio web para no bombardear el sitio con cientos de peticiones concurrentes.

```
semaforo = threading.Semaphore(4)

def descargar(url):
  semaforo.acquire()
  urllib.urlretrieve(url)
  semaforo.release()
```

Las condiciones (clase `Condition`) son de utilidad para hacer que los threads sólo puedan entrar en la sección crítica de darse una cierta condición o evento. Para esto utilizan un `Lock` pasado como parámetro, o crean un objeto `RLock` automáticamente si no se pasa ningún parámetro al constructor.

Son especialmente adecuadas para el clásico problema de productor-consumidor. La clase cuenta con métodos `acquire` y `release`, que llamarán a los métodos correspondientes del candado asociado. También tenemos métodos `wait`, `notify` y `notifyAll`.

El método `wait` debe llamarse después de haber adquirido el candado con `acquire`. Este método libera el candado y bloquea al thread hasta que una llamada a `notify` o `notifyAll` en otro thread le indican que se ha cumplido la condición por la que esperaba. El thread que informa a los demás de que se ha producido la condición, también debe llamar a `acquire` antes de llamar a `notify` o `notifyAll`.

Al llamar a `notify`, se informa del evento a un solo thread, y por tanto se despierta un

solo thread. Al llamar a `notifyAll` se despiertan todos los threads que esperaban a la condición.

Tanto el thread que notifica como los que son notificados tienen que terminar liberando el `lock` con `release`.

```
lista = []
cond = threading.Condition()

def consumir():
  cond.acquire()
  cond.wait()
  obj = lista.pop()
  cond.release()
  return obj

def producir(obj):
  cond.acquire()
  lista.append(obj)
  cond.notify()
  cond.release()
```

Los eventos, implementados mediante la clase `Event`, son un wrapper por encima de `Condition` y sirven principalmente para coordinar threads mediante señales que indican que se ha producido un evento. Los eventos nos abstraen del hecho de que estemos utilizando un `Lock` por debajo, por lo que carecen de métodos `acquire` y `release`.

El thread que debe esperar el evento llama al método `wait` y se bloquea, opcionalmente pasando como parámetro un número en coma flotante indicando el número máximo de segundos a esperar. Otro hilo, cuando ocurre el evento, manda la señal a los threads bloqueados a la espera de dicho evento utilizando el método `set`. Los threads que estaban esperando se desbloquean una vez recibida la señal. El flag que determina si se ha producido el evento se puede volver a establecer a falso usando `clear`.

Como vemos los eventos son muy similares a las condiciones, a excepción de que se desbloquean todos los threads que esperaban el evento y que no tenemos que llamar a `acquire` y `release`.

```
import threading, time

class MiThread(threading.Thread):
  def __init__(self, evento):
    threading.Thread.__init__(self)
    self.evento = evento

  def run(self):
    print self.getName(), “esperando al evento”
    self.evento.wait()
    print self.getName(), “termina la espera”

evento = threading.Event()
t1 = MiThread(evento)
t1.start()
t2 = MiThread(evento)
t2.start()
```

```
# Esperamos un poco
time.sleep(5)
evento.set()
```

Por último, un pequeño extra. Si sois usuarios de Java sin duda estaréis echando en falta una palabra clave `syncronized` para hacer que sólo un thread pueda acceder al método sobre el que se utiliza a la vez. Una construcción común es el uso de un decorador para implementar esta funcionalidad usando un `Lock`. Sería algo así:

```
def synchronized(lock):
  def dec(f):
    def func_dec(*args, **kwargs):
      lock.acquire()
      try:
        return f(*args, **kwargs)
      finally:
        lock.release()
    return func_dec
  return dec

class MyThread(threading.Thread):
  @synchronized(mi_lock)
  def run(self):
    print “metodo sincronizado”
```

Datos globales independientes

Como ya hemos comentado los threads comparten las variables globales. Sin embargo pueden existir situaciones en las que queramos utilizar variables globales pero que estas variables se comporten como si fueran locales a un solo thread. Es decir, que cada uno de los threads tengan valores distintos independientes, y que los cambios de un determinado thread sobre el valor no se vean reflejados en las copias de los demás threads.

Para lograr este comportamiento se puede utilizar la clase `threading.local`, que crea un almacén de datos locales. Primero debemos crear una instancia de la clase, o de una subclase, para después almacenar y obtener los valores a través de parámetros de la clase.

```
datos_locales = threading.local()
datos_locales.mi_var = “hola”
print datos_locales.mi_var
```

Fijémonos en el siguiente código, por ejemplo. Para el hilo principal el objeto local tiene un atributo `var`, y por lo tanto el `print` imprime su valor sin problemas. Sin embargo para el hilo `t` ese atributo no existe, y por lo tanto lanza una excepción.

```
local = threading.local()

def f():
  print local.var

local.var = “hola”
t = threading.Thread(target=f)
print local.var
t.start()
t.join()
```

Compartir información

Para compartir información entre los threads de forma sencilla podemos utilizar la clase `Queue.Queue`, que implementa una cola (una estructura de datos de tipo FIFO) con soporte multihilo. Esta clase utiliza las primitivas de `threading` para ahorrarnos tener que sincronizar el acceso a los datos nosotros mismos.

El constructor de `Queue` toma un parámetro opcional indicando el tamaño máximo de la cola. Si no se indica ningún valor no hay límite de tamaño.

Para añadir un elemento a la cola se utiliza el método `put(item)`; para obtener el siguiente elemento, `get()`. Ambos métodos tienen un parámetro booleano opcional `block` que indica si queremos que se espere hasta que haya algún elemento en la cola para poder devolverlo o hasta que la cola deje de estar llena para poder introducirlo.

También existe un parámetro opcional `timeout` que indica, en segundos, el tiempo máximo a esperar. Si el `timeout` acaba sin poder haber realizado la operación debido a que la cola estaba llena o vacía, o bien si `block` era `False`, se lanzará una excepción de tipo `Queue.Full` o `Queue.Empty`, respectivamente.

Con `qsize` obtenemos el tamaño de la cola y con `empty()` y `full()` podemos comprobar si está vacía o llena.

```
q = Queue.Queue()

class MiThread(threading.Thread):
  def __init__(self, q):
    self.q = q
    threading.Thread.__init__(self)

  def run(self):
    while True:
      try:
        obj = q.get(False)
      except Queue.Empty:
        print “Fin”
        break
      print obj

for i in range(10):
  q.put(i)

t = MiThread(q)
t.start()
t.join()
```

SERIALIZACIÓN DE OBJETOS

Algunas veces tenemos la necesidad de guardar un objeto a disco para poder recuperarlo más tarde, o puede que nos sea necesario mandar un objeto a través de la red, a otro programa en Python ejecutándose en otra máquina.

Al proceso de transformar el estado de un objeto en un formato que se pueda almacenar, recuperar y transportar se le conoce con el nombre de serialización o *marshalling*.

En Python tenemos varios módulos que nos facilitan esta tarea, como `marshal`, `pickle`, `cPickle` y `shelve`.

El módulo `marshal` es el más básico y el más primitivo de los tres, y es que, de hecho, su propósito principal y su razón de ser no es el de serializar objetos, sino trabajar con bytecode Python (archivos .pyc).

`marshal` sólo permite serializar objetos simples (la mayoría de los tipos incluidos por defecto en Python), y no proporciona ningún tipo de mecanismo de seguridad ni comprobaciones frente a datos corruptos o mal formateados. Es más, el formato utilizado para guardar el bytecode (y por tanto el formato utilizado para guardar los objetos con `marshal`) puede cambiar entre versiones, por lo que no es adecuado para almacenar datos de larga duración.

`pickle`, por su parte, permite serializar casi cualquier objeto (objetos de tipos definidos por el usuario, colecciones que contienen colecciones, etc) y cuenta con algunos mecanismos de seguridad básicos. Sin embargo, al ser más complejo que `marshal`, y, sobre todo, al estar escrito en Python en lugar de en C, como `marshal`, también es mucho más lento.

La solución, si la velocidad de la serialización es importante para nuestra aplicación, es utilizar `cPickle`, que no es más que es una implementación en C de `pickle`. `cPickle` es hasta 1000 veces más rápido que `pickle`, y prácticamente igual de rápido que `marshal`.

Si intentamos importar `cPickle` y se produce un error por algún motivo, se lanzará una excepción de tipo `ImportError`. Para utilizar `cPickle` si está disponible y `pickle` en caso contrario, podríamos usar un código similar al siguiente:

```
try:
  import cPickle as pickle
except ImportError:
  import pickle
```

`as` en un `import` sirve para importar el elemento seleccionado utilizando otro nombre indicado, en lugar de su nombre.

La forma más sencilla de serializar un objeto usando `pickle` es mediante una llamada a la función `dump` pasando como argumento el objeto a serializar y un objeto archivo en el que guardarlo (o cualquier otro tipo de objeto similar a un archivo, siempre que ofrezca métodos `read`, `readline` y `write`).

```
try:
  import cPickle as pickle
except ImportError:
  import pickle

fichero = file(“datos.dat”, “w”)
animales = [“piton”, “mono”, “camello”]

pickle.dump(animales, fichero)

fichero.close()
```

La función `dump` también tiene un parámetro opcional `protocol` que indica el protocolo a utilizar al guardar. Por defecto su valor es 0, que utiliza formato texto y es el menos eficiente. El protocolo 1 es más eficiente que el 0, pero menos que el 2. Tanto el protocolo 1 como el 2 utilizan un formato binario para guardar los datos.

```
try:
  import cPickle as pickle
except ImportError:
  import pickle

fichero = file(“datos.dat”, “w”)
animales = [“piton”, “mono”, “camello”]

pickle.dump(animales, fichero, 2)

fichero.close()
```

Para volver a cargar un objeto serializado se utiliza la función `load`, a la que se le pasa el archivo en el que se guardó.

```
try:
  import cPickle as pickle
except ImportError:
  import pickle

fichero = file(“datos.dat”, “w”)
animales = [“piton”, “mono”, “camello”]

pickle.dump(animales, fichero)

fichero.close()

fichero = file(“datos.dat”)

animales2 = pickle.load(fichero)
print animales2
```

Supongamos ahora que queremos almacenar un par de listas en un fichero. Esto sería tan sencillo como llamar una vez a `dump` por cada lista, y llamar después una vez a `load` por cada lista.

```
fichero = file(“datos.dat”, “w”)
animales = [“piton”, “mono”, “camello”]
lenguajes = [“python”, “mono”, “perl”]

pickle.dump(animales, fichero)
```

```
pickle.dump(lenguajes, fichero)

fichero = file("datos.dat")

animales2 = pickle.load(fichero)
lenguajes2 = pickle.load(fichero)
print animales2
print lenguajes2
```

Pero, ¿y si hubiéramos guardado 30 objetos y quisiéramos acceder al último de ellos? ¿o si no recordáramos en qué posición lo habíamos guardado? El módulo `shelve` extiende `pickle` / `cPickle` para proporcionar una forma de realizar la serialización más clara y sencilla, en la que podemos acceder a la versión serializada de un objeto mediante una cadena asociada, a través de una estructura parecida a un diccionario.

La única función que necesitamos conocer del módulo `shelve` es `open`, que cuenta con un parámetro `filename` mediante el que indicar la ruta a un archivo en el que guardar los objetos (en realidad se puede crear más de un archivo, con nombres basados en `filename`, pero esto es transparente al usuario).

La función `open` también cuenta con un parámetro opcional `protocol`, con el que especificar el protocolo que queremos que utilice `pickle` por debajo.

Como resultado de la llamada a `open` obtenemos un objeto `Shelf`, con el que podemos trabajar como si de un diccionario normal se tratase (a excepción de que las claves sólo pueden ser cadenas) para almacenar y recuperar nuestros objetos.

Como un diccionario cualquiera la clase `Shelf` cuenta con métodos `get`, `has_key`, `items`, `keys`, `values`, …

Una vez hayamos terminado de trabajar con el objeto `Shelf`, lo cerraremos utilizando el método `close`.

```
import shelve

animales = ["piton", "mono", "camello"]
lenguajes = ["python", "mono", "perl"]

shelf = shelve.open("datos.dat")
shelf["primera"] = animales
shelf["segunda"] = lenguajes

print shelf["segunda"]

shelf.close()
```

BASES DE DATOS

Existen problemas para los que guardar nuestros datos en ficheros de texto plano, en archivos XML, o mediante serialización con `pickle` o `shelve` pueden ser soluciones poco convenientes. En ocasiones no queda más remedio que recurrir a las bases de datos, ya sea por cuestiones de escalabilidad, de interoperabilidad, de coherencia, de seguridad, de confidencialidad, etc.

A lo largo de este capítulo aprenderemos a trabajar con bases de datos en Python. Sin embargo se asumen una serie de conocimientos básicos, como puede ser el manejo elemental de SQL. Si este no es el caso, existen miles de recursos a disposición del lector en Internet para introducirse en el manejo de bases de datos.

DB API

Existen cientos de bases de datos en el mercado, tanto comerciales como gratuitas. También existen decenas de módulos distintos para trabajar con dichas bases de datos en Python, lo que significa decenas de APIs distintas por aprender.

En Python, como en otros lenguajes como Java con JDBC, existe una propuesta de API estándar para el manejo de bases de datos, de forma que el código sea prácticamente igual independientemente de la base de datos que estemos utilizando por debajo. Esta especificación recibe el nombre de Python Database API o DB-API y se recoge en el PEP 249 (http://www.python.org/dev/peps/pep-0249/).

DB-API se encuentra en estos momentos en su versión 2.0, y existen implementaciones para las bases de datos relacionales más conocidas, así como para algunas bases de datos no relacionales.

A lo largo de este capítulo utilizaremos la base de datos SQLite para los ejemplos, ya que no se necesita instalar y ejecutar un proceso servidor independiente con el que se comunique el programa, sino que se trata de una pequeña librería en C que se integra con la aplicación y que viene incluida con Python por defecto desde la versión 2.5. Desde la misma versión Python también incorpora un módulo compatible con esta base de datos que sigue la especificación de DB API 2.0: `sqlite3`, por lo que no necesitaremos ningún tipo de configuración extra.

Nada impide al lector, no obstante, instalar y utilizar cualquier otra base de datos, como MySQL, con la cual podemos trabajar a través del driver compatible con DB API 2.0 `MySQLdb` (http://mysql-python.sourceforge.net/).

Variables globales

Antes de comenzar a trabajar con sqlite3, vamos a consultar algunos datos interesantes sobre el módulo. Todos los drivers compatibles con DB-API 2.0 deben tener 3 variables globales que los describen. A saber:

- `apilevel`: una cadena con la versión de DB API que utiliza. Actualmente sólo puede tomar como valor “1.0” o “2.0”. Si la variable no existe se asume que es 1.0.
- `threadsafety`: se trata de un entero de 0 a 3 que describe lo seguro que es el módulo para el uso con threads. Si es 0 no se puede compartir el módulo entre threads sin utilizar algún tipo de mecanismo de sincronización; si es 1, pueden compartir el módulo pero no las conexiones; si es 2, módulos y conexiones pero no cursores y, por último, si es 3, es totalmente *thread-safe*.
- `paramstyle`: informa sobre la sintaxis a utilizar para insertar valores en la

consulta SQL de forma dinámica.

- `qmark`: interrogaciones.

  ```
  sql = “select all from t where valor=?”
  ```

- `numeric`: un número indicando la posición.

  ```
  sql = “select all from t where valor=:1”
  ```

- `named`: el nombre del valor.

  ```
  sql = “select all from t where valor=:valor”
  ```

- `format`: especificadores de formato similares a los del `printf` de C.

  ```
  sql = “select all from t where valor=%s”
  ```

- `pyformat`: similar al anterior, pero con las extensiones de Python.

  ```
  sql = “select all from t where valor=%(valor)”
  ```

Veamos los valores correspondientes a sqlite3:

```
>>> import sqlite3 as dbapi
>>> print dbapi.apilevel
2.0
>>> print dbapi.threadsafety
1
>>> print dbapi.paramstyle
qmark
```

Excepciones

A continuación podéis encontrar la jerarquía de excepciones que deben proporcionar los módulos, junto con una pequeña descripción de cada excepción, a modo de referencia.

```
StandardError
|__Warning
|__Error
  |__InterfaceError
  |__DatabaseError
    |__DataError
    |__OperationalError
    |__IntegrityError
    |__InternalError
    |__ProgrammingError
    |__NotSupportedError
```

- `StandardError`: Super clase para todas las excepciones de DB API.
- `Warning`: Excepción que se lanza para avisos importantes.
- `Error`: Super clase de los errores.
- `InterfaceError`: Errores relacionados con la interfaz de la base de datos, y no con la base de datos en sí.
- `DatabaseError`: Errores relacionados con la base de datos.

- `DataError`: Errores relacionados con los datos, como una división entre cero.
- `OperationalError`: Errores relacionados con el funcionamiento de la base de datos, como una desconexión inesperada.
- `IntegrityError`: Errores relacionados con la integridad referencial.
- `InternalError`: Error interno de la base de datos.
- `ProgrammingError`: Errores de programación, como errores en el código SQL.
- `NotSupportedError`: Excepción que se lanza cuando se solicita un método que no está soportado por la base de datos.

Uso básico de DB-API

Pasemos ahora a ver cómo trabajar con nuestra base de datos a través de DB-API.

Lo primero que tendremos que hacer es realizar una conexión con el servidor de la base de datos. Esto se hace mediante la función `connect`, cuyos parámetros no están estandarizados y dependen de la base de datos a la que estemos conectándonos.

En el caso de sqlite3 sólo necesitamos pasar como parámetro una cadena con la ruta al archivo en el que guardar los datos de la base de datos, o bien la cadena `“:memory:”` para utilizar la memoria RAM en lugar de un fichero en disco.

Por otro lado, en el caso de `MySQLdb`, `connect` toma como parámetros la máquina en la que corre el servidor (`host`), el puerto (`port`), nombre de usuario con el que autenticarse (`user`), contraseña (`password`) y base de datos a la que conectarnos de entre las que se encuentran en nuestro SGBD (`db`).

La función `connect` devuelve un objeto de tipo `Connection` que representa la conexión con el servidor.

```
>>> bbdd = dbapi.connect(“bbdd.dat”)
>>> print bbdd
<sqlite3.Connection object at 0x00A71DA0>
```

Las distintas operaciones que podemos realizar con la base de datos se realizan a través de un objeto `Cursor`. Para crear este objeto se utiliza el método `cursor()` del objeto `Connection`:

```
c = bbdd.cursor()
```

Las operaciones se ejecutan a través del método `execute` de `Cursor`, pasando como parámetro una cadena con el código SQL a ejecutar.

Como ejemplo creemos una nueva tabla empleados en la base de datos:

```
c.execute(“““create table empleados (dni text,
     nombre text,
     departamento text)”””)
```

y a continuación, insertemos una tupla en nuestra nueva tabla:

```
c.execute(“““insert into empleados
     values (‘12345678-A’, ‘Manuel Gil’, ‘Contabilidad’)”””)
```

Si nuestra base de datos soporta transacciones, si estas están activadas, y si la característica de *auto-commit* está desactivada, será necesario llamar al método `commit` de la conexión para que se lleven a cabo las operaciones definidas en la transacción.

Si en estas circunstancias utilizáramos una herramienta externa para comprobar el contenido de nuestra base de datos sin hacer primero el `commit` nos encontraríamos entonces con una base de datos vacía.

Si comprobáramos el contenido de la base de datos desde Python, sin cerrar el cursor ni la conexión, recibiríamos el resultado del contexto de la transacción, por lo que parecería que se han llevado a cabo los cambios, aunque no es así, y los cambios sólo se aplican, como comentamos, al llamar a `commit`.

Para bases de datos que no soporten transacciones el estándar dicta que debe proporcionarse un método `commit` con implementación vacía, por lo que no es mala idea llamar siempre a `commit` aunque no sea necesario para poder cambiar de sistema de base de datos con solo modificar la línea del `import`.

Si nuestra base de datos soporta la característica de *rollback* también podemos cancelar la transacción actual con:

```
bbdd.rollback()
```

Si la base de datos no soporta rollback llamar a este método producirá una excepción.

Veamos ahora un ejemplo completo de uso:

```
import sqlite3 as dbapi

bbdd = dbapi.connect(“bbdd.dat”)
cursor = bbdd.cursor()

cursor.execute(“““create table empleados (dni text,
     nombre text,
     departamento text)”””)

cursor.execute(“““insert into empleados
     values (‘12345678-A’, ‘Manuel Gil’, ‘Contabilidad’)”””)

bbdd.commit()

cursor.execute(“““select * from empleados
     where departamento=’Contabilidad’”””)

for tupla in cursor.fetchall():
  print tupla
```

Como vemos, para realizar consultas a la base de datos también se utiliza `execute`. Para consultar las tuplas resultantes de la sentencia SQL se puede llamar a los métodos de `Cursor` `fetchone`, `fetchmany` o `fetchall` o usar el objeto `Cursor` como un iterador.

```
cursor.execute("""select * from empleados
     where departamento='Contabilidad'""")

for resultado in cursor:
  print tupla
```

El método `fetchone` devuelve la siguiente tupla del conjunto resultado o `None` cuando no existen más tuplas, `fetchmany` devuelve el número de tuplas indicado por el entero pasado como parámetro o bien el número indicado por el atributo `Cursor.arraysize` si no se pasa ningún parámetro (`Cursor.arraysize` vale 1 por defecto) y `fetchall` devuelve un objeto iterable con todas las tuplas.

A la hora de trabajar con selects u otros tipos de sentencias SQL es importante tener en cuenta que no deberían usarse los métodos de cadena habituales para construir las sentencias, dado que esto nos haría vulnerables a ataques de inyección SQL, sino que en su lugar debe usarse la característica de sustitución de parámetros de DB API.

Supongamos que estamos desarrollando una aplicación web con Python para un banco y que se pudiera consultar una lista de sucursales del banco en una ciudad determinada con una URL de la forma http://www.mibanco.com/sucursales?ciudad=Madrid

Podríamos tener una consulta como esta:

```
cursor.execute("""select * from sucursales
     where ciudad='" + ciudad + "'""")
```

A primera vista podría parecer que no existe ningún problema: no hacemos más que obtener las sucursales que se encuentren en la ciudad indicada por la variable ciudad. Pero, ¿qué ocurriría si un usuario malintencionado accediera a una URL como "http://www.mibanco.com/sucursales?ciudad=Madrid';SELECT * FROM contrasenyas"?

Como no se realiza ninguna validación sobre los valores que puede contener la variable ciudad, sería sencillo que alguien pudiera hacerse con el control total de la aplicación.

Lo correcto sería, como decíamos, utilizar la característica de sustitución de parámetros de DB API. El valor de `paramstyle` para el módulo sqlite3 era `qmark`. Esto significa que debemos escribir un signo de interrogación en el lugar en el que queramos insertar el valor, y basta pasar un segundo parámetro a `execute` en forma de secuencia o `mapping` con los valores a utilizar para que el módulo cree la sentencia por nosotros.

```
cursor.execute("""select * from sucursales
     where ciudad=?""", (ciudad,))
```

Por último, al final del programa se debe cerrar el cursor y la conexión:

```
cursor.close()
bbdd.close()
```

Tipos SQL

En ocasiones podemos necesitar trabajar con tipos de SQL, y almacenar, por ejemplo, fechas u horas usando `Date` y `Time` y no con cadenas. La API de bases de datos de Python incluye una serie de constructores a utilizar para crear estos tipos. Estos son:

- `Date(year, month, day)`: Para almacenar fechas.
- `Time(hour, minute, second)`: Para almacenar horas.
- `Timestamp(year, month, day, hour, minute, second)`: Para almacenar `timestamps` (una fecha con su hora).
- `DateFromTicks(ticks)`: Para crear una fecha a partir de un número con los segundos transcurridos desde el *epoch* (el 1 de Enero de 1970 a las 00:00:00 GMT).
- `TimeFromTicks(ticks)`: Similar al anterior, para horas en lugar de fechas.
- `TimestampFromTicks(ticks)`: Similar al anterior, para timestamps.
- `Binary(string)`: Valor binario.

Otras opciones

Por supuesto no estamos obligados a utilizar DB-API, ni bases de datos relacionales. En Python existen módulos para trabajar con bases de datos orientadas a objetos, como ZODB (Zope Object Database) y motores para mapeo objeto-relacional (ORM) como SQLAlchemy, SQLObject o Storm.

Además, si utilizamos IronPython en lugar de CPython tenemos la posibilidad de utilizar las conexiones a bases de datos de .NET, y si utilizamos Jython, las de Java.

DOCUMENTACIÓN

Docstrings

En capítulos anteriores ya comentamos en varias ocasiones que todos los objetos cuentan con una variable especial `__doc__` mediante la que indicar el propósito y uso del objeto. Estos son los llamados *docstrings* o cadenas de documentación.

A estos atributos se les puede asociar el texto correspondiente explícitamente, asignándolo al literal cadena correspondiente, como con cualquier otra variable. Sin embargo, por conveniencia, Python ofrece un mecanismo mucho más sencillo y es que si el primer estamento de la definición del objeto es una cadena, esta se asocia a la variable `__doc__` automáticamente.

```
def haz_algo(arg):
  """Este es el docstring de la funcion."""
  print arg

print haz_algo.__doc__

haz_algo.__doc__ = """Este es un nuevo docstring."""

print haz_algo.__doc__
```

Como vemos lo interesante de estas cadenas es que, a diferencia de los comentarios normales de Python y de los comentarios de otros lenguajes, las cadenas de documentación no se eliminan del bytecode, por lo que se pueden consultar en tiempo de ejecución, usando, por ejemplo, la función `help` del lenguaje, o utilizando la sentencia `print` como en el ejemplo anterior.

```
>>> help(haz_algo)
Help on function haz_algo in module __main__:
haz_algo(arg)
  Este es un nuevo docstring.
```

Pydoc

La función `help`, que comentamos brevemente con anterioridad, utiliza el módulo `pydoc` para generar la documentación de un objeto a partir de su docstring y los docstrings de sus miembros. Este módulo, incluido por defecto con Python desde la versión 2.1, se puede importar en nuestro código Python y utilizarse programáticamente, o bien se puede utilizar como una herramienta de línea de comandos que sería el equivalente a la aplicación Javadoc del mundo Java.

`pydoc` puede mostrar la información como texto en la consola, tal como lo utiliza `help`, pero también puede generar archivos HTML como javadoc o facilitar la información a través de un pequeño servidor web incluido con el módulo.

Pydoc es muy sencillo de utilizar. Con

```
pydoc.py nombre1 [nombre2 ...]
```

se muestra la documentación del tema, módulo, clase, paquete, función o palabra clave indicada de forma similar a la función `help`. Si el nombre es `keywords`, `topics` o `modules` se listarán las distintas palabras claves, temas y módulos respectivamente.

Si se pasa el flag `-w`, el script guardará la documentación en uno o varios archivos html en lugar de mostrarla por pantalla.

```
pydoc.py -w nombre1 [nombre2 ...]
```

El flag `-k` sirve para buscar una determinada palabra en las sinopsis de todos los módulos disponibles. La sinopsis es la primera línea de la cadena de documentación.

```
pydoc.py -k xml
```

Con `-p` podemos iniciar el servidor HTTP en el puerto indicado.

```
pydoc.py -p puerto
```

Una vez hecho esto podemos acceder a la documentación de todos los módulos disponibles abriendo la página *http://localhost:puerto* en nuestro navegador.

Por último, mediante el flag `-g` podemos lanzar una interfaz gráfica para buscar documentación que utiliza el servidor HTTP para mostrar los resultados.

Epydoc y reStructuredText

El problema de `pydoc` es que es muy simple, y no permite añadir semántica o modificar estilos de la documentación. No podemos, por ejemplo, indicar que en una línea en concreto de entre las líneas de documentación de la función describe un parámetro de la función o mostrar un cierto término en cursiva.

Existen proyectos para generar documentación con funcionalidades más avanzadas como Docutils, Epydoc o Sphinx, aunque es necesario aprender sintaxis especiales.

Docutils es un proyecto desarrollado por David Goodger que incluye distintas herramientas para generar documentación utilizando el formato reStructuredText, un formato de texto plano creado por el mismo autor, y que es el formato más utilizado en la comunidad Python. reStructuredText se utiliza, entre otros, para la creación de los PEPs (*Python Enhancement Proposals*).

Sin embargo, actualmente Docutils es más indicado para generar documentos a partir de archivos de texto, y no a partir de docstrings extraídos de código fuente Python, ya que el parser encargado de este trabajo dista mucho de estar terminado.

EpyDoc es una de las herramientas de generación de documentación para Python más utilizadas. Además de texto plano y de su propio formato, llamado epytext, soporta reStructuredText y sintaxis Javadoc, cosa que los programadores Java agradecerán.

A lo largo del resto del capítulo utilizaremos reStructuredText como lenguaje de marcado y EpyDoc para generar los documentos finales.

Epydoc se puede descargar desde su página web en forma de instalador exe para Windows, paquete RPM para Fedora o similares, o en archivos zip y tar.gz que incluyen scripts de instalación: *http://epydoc.sourceforge.net/*. También se encuentra en los repositorios de varias distribuciones Linux.

Una vez hayamos instalado Epydoc siguiendo el método adecuado para nuestro sistema operativo tendremos acceso a su funcionalidad a través de dos interfaces de usuario distintas: el script `epydoc`, que consiste en una aplicación de línea de comandos, y el script `epydocgui` (`epydoc.pyw` en Windows), que ofrece una interfaz gráfica. Además también podemos acceder a la funcionalidad de `epydoc` programáticamente, como en el caso de `pydoc`.

Vamos a crear un pequeño módulo con un par de clases para ver primero el resultado de utilizar `epydoc` con docstrings de texto plano, sin ningún tipo de marcado especial.

```
"""Modulo para ejemplificar el uso de epydoc."""

class Persona:
  """Mi clase de ejemplo."""
  def __init__(self, nombre):
    """Inicializador de la clase Persona."""
```

```
    self.nombre = nombre
    self.mostrar_nombre()

  def mostrar_nombre(self):
    “““Imprime el nombre de la persona”””
    print “Esta es la persona %s” % self.nombre

class Empleado(Persona):
  “““Subclase de Persona.”””
  pass

if __name__ == “__main__”:
  raul = Persona(“Raul”)
```

El formato de salida por defecto de `epydoc` es HTML. Por lo tanto para generar la documentación en forma de documentos HTML bastaría escribir algo así:

```
epydoc ejemplo.py
```

o bien

```
epydoc --html ejemplo.py
```

Para generar un archivo PDF, utilizando LaTeX, se utilizaría el flag `--pdf`:

```
epydoc --pdf ejemplo.py
```

Si LaTeX no está instalado o epydoc no encuentra el ejecutable no será posible generar el PDF.

También podemos indicar el nombre del proyecto y la URL mediante las opciones `--name` y `--url`:

```
epydoc --name Ejemplo --url http://mundogeek.net ejemplo.py
```

E incluso añadir diagramas mostrando la clase base y subclases (`--graph classtree`), las llamadas entre funciones y métodos (`--graph callgraph`), clases y subclases usando notación UML (`--graph uml-classtree`) o todos ellos (`--graph all`).

```
epydoc --graph all ejemplo.py
```

Para generar el grafo de llamadas, no obstante, es necesario generar un archivo con la información necesaria utilizando el módulo profile o el módulo hotshot e indicar el archivo resultante utilizando el flag `--pstat`:

```
epydoc --graph all --pstat profile.out ejemplo.py
```

Veamos ahora algunas funcionalidades básicas de marcado en reStructuredText.

Para poner un texto en itálica se rodea el texto con asteriscos:

itálica -> *itálica*

Para ponerlo en negrita, se utilizan dos asteriscos:

negrita -> **negrita**

Para mostrar el texto como monoespacio, por ejemplo para mostrar código inline, se utiliza “.

```
“monoespacio” -> monoespacio
```

Si necesitamos utilizar cualquiera de estos caracteres especiales, se pueden escapar utilizando la barra invertida.

```
\* es un carácter especial -> * es un carácter especial
```

Los títulos se crean añadiendo una línea de caracteres no alfanuméricos por debajo del texto, o por encima y por debajo del texto. Para crear un subtitulo basta utilizar una nueva combinación.

```
Título
======
Subtitulo
---------
```

Título

Subtitulo

Para crear una lista no ordenada se empieza cada línea con el carácter ‘*’, ‘-’ o ‘+’:

```
* Python
* C
* Java
```

- Python
- C
- Java

Para crear una lista numerada se empieza la línea con el número seguido de un punto, o bien con el símbolo ‘#’ para que se introduzca el número automáticamente.

```
1. Python
2. C
3. Java
```

1. Python
2. C
3. Java

Para describir propiedades de los elementos que estamos documentando se utilizan los campos o fields. En reStructuredText los campos comienzan con ‘:’, le sigue el nombre del campo y opcionalmente sus argumentos, y se cierra de nuevo con ‘:’, para terminar con el cuerpo del campo.

Estos son algunos de los campos que soporta Epydoc:

Funciones y métodos	
`:param p:` Un parámetro	Describe el parámetro `p`.
`:type p: str`	Especifica el tipo esperado para el parámetro `p`.

`:return:` `True` si son iguales	Valor de retorno.
`:rtype: str`	Tipo del valor de retorno.
`:keyword p:` Un parámetro	Descripción del parámetro con valor por defecto y nombre `p`.
`:raise e:` Si el parámetro es cero	Describe las circunstancias para las que se lanza la excepción `e`.

Variables	
`:ivar v:` Una variable	Descripción de la instancia `v`.
`:cvar v:` Una variable	Descripción de la variable estática de clase `v`.
`:var v:` Una variable	Descripción de la variable `v` del módulo.
`:type v: str`	Tipo de la variable `v`.

Notas	
`:note:` Una nota	Una nota sobre el objeto.
`:attention:` Importante	Una nota importante sobre el objeto.
`:bug:` No funciona para el valor 0	Descripción de un error en el objeto.
`:warning:` Cuidado con el valor 0	Una advertencia acerca de un objeto.
`:see:` Ver ‘Python para todos’	Para indicar información relacionada.

Estado	
`:version:` 1.0	Versión actual del objeto.
`:change:` Versión inicial	Listado de cambios.
`:todo:` Internacionalización	Un cambio planeado para el objeto.
`:status:` Versión estable	Estado del objeto.

Autoría	
`:author:` Raul Gonzalez	Autor o autores del objeto.
`:organization:` Mundo geek	Organización que creó o mantiene el objeto.
`:license:` GPL	Licencia del objeto.
`:contact:` zootropo en gmail	Información de contacto del autor.

Para que Epydoc sepa que utilizamos reStructuredText es necesario indicarlo mediante una variable `__docformat__` en el código, o bien mediante la opción `--docformat` de línea de comandos. Las opciones posibles son `epytext`, `plaintext`, `restructuredtext` o `javadoc`.

Veamos un ejemplo con campos:

```
"""Modulo para ejemplificar el uso de *epydoc*.
  :author: Raul Gonzalez
  :version: 0.1"""

__docformat__ = "restructuredtext"
```

```
class Persona:
  “““Modela una persona.”””
  def __init__(self, nombre, edad):
    “““Inicializador de la clase `Persona`.
      :param nombre: Nombre de la persona.
      :param edad: Edad de la persona”””
    self.nombre = nombre
    self.edad = edad
    self.mostrar_nombre()
  def mostrar_nombre(self):
    “““Imprime el nombre de la persona”””
    print “Esta es la persona %s” % self.nombre
class Empleado(Persona):
  “““Subclase de `Persona` correspondiente a las personas
    que trabajan para la organizacion.
    :todo: Escribir implementacion.”””
  pass
if __name__ == “__main__”:
  juan = Persona(“Juan”, 26)
```

reStructuredText también soporta un segundo tipo de campos en el que el cuerpo del campo es una lista. De esta forma podemos, por ejemplo, describir todos los parámetros de una función o método con un solo campo `:Parameters:`, en lugar de con un campo `:param:` para cada parámetro.

```
class Persona:
  “““Modela una persona.”””
  def __init__(self, nombre, edad):
    “““Inicializador de la clase `Persona`.

      :Parameters:
        - `nombre`: Nombre de la persona.
        - `edad`: Edad de la persona.
    ”””

    self.nombre = nombre
    self.edad = edad
    self.mostrar_nombre()
```

Otros campos que admiten listas son `:Exceptions:` para indicar varias excepciones (`:except:`), `:Variables:` para comentar varias variables (`:var:`) o `:Ivariables:` para comentar varias instancias (`:ivar:`).

PRUEBAS

Para asegurar en la medida de lo posible el correcto funcionamiento y la calidad del software se suelen utilizar distintos tipos de pruebas, como pueden ser las pruebas unitarias, las pruebas de integración, o las pruebas de regresión.

A lo largo de este capítulo nos centraremos en las pruebas unitarias, mediante las que se comprueba el correcto funcionamiento de las unidades lógicas en las que se divide el programa, sin tener en cuenta la interrelación con otras unidades.

La solución más extendida para las pruebas unitarias en el mundo Python es `unittest`, a menudo combinado con `doctest` para pruebas más sencillas. Ambos módulos están incluidos en la librería estándar de Python.

Doctest

Como es de suponer por el nombre del módulo, `doctest` permite combinar las pruebas con la documentación. Esta idea de utilizar las pruebas unitarias para probar el código y también a modo de documentación permite realizar pruebas de forma muy sencilla, propicia el que las pruebas se mantengan actualizadas, y sirve a modo de ejemplo de uso del código y como ayuda para entender su propósito.

Cuando `doctest` encuentra una línea en la documentación que comienza con '>>>' se asume que lo que le sigue es código Python a ejecutar, y que la respuesta esperada se encuentra en la línea o líneas siguientes, sin >>>. El texto de la prueba termina cuando se encuentra una línea en blanco, o cuando se llega al final de la cadena de documentación.

Tomemos como ejemplo la siguiente función, que devuelve una lista con los cuadrados de todos los números que componen la lista pasada como parámetro:

```
def cuadrados(lista):
  """Calcula el cuadrado de los numeros de una lista"""

  return [n ** 2 for n in lista]
```

Podríamos crear una prueba como la siguiente, en la que comprobamos que el resultado al pasar la lista `[0, 1, 2, 3]` es el que esperábamos:

```
def cuadrados(lista):
  """Calcula el cuadrado de los numeros de una lista

  >>> l = [0, 1, 2, 3]
  >>> cuadrados(l)
  [0, 1, 4, 9]
  """

  return [n ** 2 for n in lista]
```

Lo que hacemos en este ejemplo es indicar a doctest que cree un lista `l` con valor `[0, 1, 2, 3]`, que llame a continuación a la función `cuadrados` con `l` como argumento, y que compruebe que el resultado devuelto sea igual a `[0, 1, 4, 9]`.

Para ejecutar las pruebas se utiliza la función `testmod` del módulo, a la que se le puede pasar opcionalmente el nombre de un módulo a evaluar (parámetro `name`). En el caso de que no se indique ningún argumento, como en este caso, se evalúa el módulo actual:

```
def cuadrados(lista):
  """Calcula el cuadrado de los numeros de una lista

  >>> l = [0, 1, 2, 3]
  >>> cuadrados(l)
  [0, 1, 4, 9]
  """

  return [n ** 2 for n in lista]

def _test():
```

```
  import doctest
  doctest.testmod()

if __name__ == “__main__”:
  _test()
```

En el caso de que el código no pase alguna de las pruebas que hemos definido, `doctest` mostrará el resultado obtenido y el resultado esperado. En caso contrario, si todo es correcto, no se mostrará ningún mensaje, a menos que añadamos la opción `-v` al llamar al script o el parámetro `verbose=True` a la función `testmod`, en cuyo caso se mostrarán todas las pruebas ejecutadas, independientemente de si se ejecutaron con éxito.

Este sería el aspecto de la salida de doctest utilizando el parámetro `-v`:

```
Trying:
  l = [0, 1, 2, 3]
Expecting nothing
ok
Trying:
  cuadrados(l)
Expecting:
  [0, 1, 4, 9]
ok
2 items had no tests:
  __main__
  __main__._test
1 items passed all tests:
  2 tests in __main__.cuadrados
2 tests in 3 items.
2 passed and 0 failed.
Test passed.
```

Ahora vamos a introducir un error en el código de la función para ver el aspecto de un mensaje de error de `doctest`. Supongamos, por ejemplo, que hubiéramos escrito un operador de multiplicación (‘`*`’) en lugar de uno de exponenciación (‘`**`’):

```
def cuadrados(lista):
  “““Calcula el cuadrado de los numeros de una lista

  >>> l = [0, 1, 2, 3]
  >>> cuadrados(l)
  [0, 1, 4, 9]
  “““

  return [n * 2 for n in lista]

def _test():
  import doctest
  doctest.testmod()

if __name__ == “__main__”:
  _test()
```

Obtendríamos algo parecido a esto:

```
****************************************
File “ejemplo.py”, line 5, in __main__.cuadrados
Failed example:
  cuadrados(l)
Expected:
  [0, 1, 4, 9]
Got:
```

```
  [0, 2, 4, 6]
**********************************************************************
1 items had failures:
  1 of 2 in __main__.cuadrados
***Test Failed*** 1 failures.
```

Como vemos, el mensaje nos indica que ha fallado la prueba de la línea 5, al llamar a `cuadrados(l)`, cuyo resultado debería ser `[0, 1, 4, 9]`, y sin embargo obtuvimos `[0, 2, 4, 6]`.

Veamos por último cómo utilizar sentencias anidadas para hacer cosas un poco más complicadas con `doctest`. En el ejemplo siguiente nuestra función calcula el cuadrado de un único número pasado como parámetro, y diseñamos una prueba que compruebe que el resultado es el adecuado para varias llamadas con distintos valores. Las sentencias anidadas comienzan con "..." en lugar de ">>>":

```
def cuadrado(num):
  """Calcula el cuadrado de un numero.

  >>> l = [0, 1, 2, 3]
  >>> for n in l:
  ... cuadrado(n)
  [0, 1, 4, 9]
  """

  return num ** 2

def _test():
  import doctest
  doctest.testmod()

if __name__ == "__main__":
  _test()
```

unittest / PyUnit

`unittest`, también llamado PyUnit, forma parte de una familia de herramientas conocida colectivamente como xUnit, un conjunto de frameworks basados en el software SUnit para Smalltalk, creado por Kent Beck, uno de los padres de la eXtreme Programming. Otros ejemplos de herramientas que forman parte de esta familia son JUnit para Java, creada por el propio Kent Beck junto a Erich Gamma, o NUnit, para .NET.

El uso de `unittest` es muy sencillo. Para cada grupo de pruebas tenemos que crear una clase que herede de `unittest.TestCase`, y añadir una serie de métodos que comiencen con test, que serán cada una de las pruebas que queremos ejecutar dentro de esa batería de pruebas.

Para ejecutar las pruebas, basta llamar a la función `main()` del módulo, con lo que se ejecutarán todos los métodos cuyo nombre comience con test, en orden alfanumérico. Al ejecutar cada una de las pruebas el resultado puede ser:

- `OK`: La prueba ha pasado con éxito.
- `FAIL`: La prueba no ha pasado con éxito. Se lanza una excepción `AssertionError` para indicarlo.
- `ERROR`: Al ejecutar la prueba se lanzó una excepción distinta de `AssertionError`.

En el siguiente ejemplo, dado que el método que modela nuestra prueba no lanza ninguna excepción, la prueba pasaría con éxito.

```
import unittest

class EjemploPruebas(unittest.TestCase):
  def test(self):
    pass

if __name__ == “__main__”:
  unittest.main()
```

En este otro, sin embargo, fallaría:

```
import unittest

class EjemploPruebas(unittest.TestCase):
  def test(self):
    raise AssertionError()

if __name__ == “__main__”:
  unittest.main()
```

Nada nos impide utilizar cláusulas `if` para evaluar las condiciones que nos interesen y lanzar una excepción de tipo `AssertionError` cuando no sea así, pero la clase `TestCase` cuenta con varios métodos que nos pueden facilitar la tarea de realizar comprobaciones sencillas. Son los siguientes:

- `assertAlmostEqual(first, second, places=7, msg=None)`: Comprueba que los objetos pasados como parámetros sean iguales hasta el séptimo decimal (o el número de decimales indicado por `places`).
- `assertEqual(first, second, msg=None)`: Comprueba que los objetos pasados como parámetros sean iguales.
- `assertFalse(expr, msg=None)`: Comprueba que la expresión sea falsa.
- `assertNotAlmostEqual(first, second, places=7, msg=None)`: Comprueba que los objetos pasados como parámetros no sean iguales hasta el séptimo decimal (o hasta el número de decimales indicado por `places`).
- `assertNotEqual(first, second, msg=None)`: Comprueba que los objetos pasados como parámetros no sean iguales.
- `assertRaises(excClass, callableObj, *args, **kwargs)`: Comprueba que al llamar al objeto `callableObj` con los parámetros definidos por `*args` y `**kwargs` se lanza una excepción de tipo `excClass`.
- `assertTrue(expr, msg=None)`: Comprueba que la expresión sea cierta.
- `assert_(expr, msg=None)`: Comprueba que la expresión sea cierta.
- `fail(msg=None)`: Falla inmediatamente.
- `failIf(expr, msg=None)`: Falla si la expresión es cierta.
- `failIfAlmostEqual(first, second, places=7, msg=None)`: Falla si los objetos pasados como parámetros son iguales hasta el séptimo decimal (o hasta el número de decimales indicado por `places`).
- `failIfEqual(first, second, msg=None)`: Falla si los objetos pasados como parámetros son iguales.
- `failUnless(expr, msg=None)`: Falla a menos que la expresión sea cierta.
- `failUnlessAlmostEqual(first, second, places=7, msg=None)`: Falla a menos que los objetos pasados como parámetros sean iguales hasta el séptimo decimal (o hasta el número de decimales indicado por `places`).
- `failUnlessEqual(first, second, msg=None)`: Falla a menos que los objetos pasados como parámetros sean iguales.
- `failUnlessRaises(excClass, callableObj, *args, **kwargs)`: Falla a menos que al llamar al objeto `callableObj` con los parámetros definidos por `*args` y `**kwargs` se lance una excepción de tipo `excClass`.

Como vemos todos los métodos cuentan con un parámetro opcional `msg` con un mensaje a mostrar cuando dicha comprobación falle.

Retomemos nuestra pequeña función para calcular el cuadrado de un número. Para probar el funcionamiento de la función podríamos hacer, por ejemplo, algo así:

```
import unittest

def cuadrado(num):
  """Calcula el cuadrado de un numero."""

  return num ** 2

class EjemploPruebas(unittest.TestCase):
  def test(self):
    l = [0, 1, 2, 3]
    r = [cuadrado(n) for n in l]
    self.assertEqual(r, [0, 1, 4, 9])

if __name__ == "__main__":
  unittest.main()
```

Preparación del contexto

En ocasiones es necesario preparar el entorno en el que queremos que se ejecuten las pruebas. Por ejemplo, puede ser necesario introducir unos valores por defecto en una base de datos, crear una conexión con una máquina, crear algún archivo, etc. Esto es lo que se conoce en el mundo de xUnit como *test fixture*.

La clase `TestCase` proporciona un par de métodos que podemos sobreescribir para construir y desconstruir el entorno y que se ejecutan antes y después de las pruebas definidas en esa clase. Estos métodos son `setUp()` y `tearDown()`.

```
class EjemploFixture(unittest.TestCase):
  def setUp(self):
    print "Preparando contexto"
    self.lista = [0, 1, 2, 3]

  def test(self):
    print "Ejecutando prueba"
    r = [cuadrado(n) for n in self.lista]
    self.assertEqual(r, [0, 1, 4, 9])

  def tearDown(self):
    print "Desconstruyendo contexto"
    del self.lista
```

DISTRIBUIR APLICACIONES PYTHON

Una vez terminemos con el desarrollo de nuestra nueva aplicación es conveniente empaquetarla de forma que sea sencillo para los usuarios instalarla, y para nosotros distribuirla.

En Python existen dos módulos principales para este cometido: `distutils`, que es parte de la librería estándar y era el método más utilizado hasta hace poco, y `setuptools`, que extiende la funcionalidad de `distutils` y es cada vez más popular.

En este capítulo veremos el funcionamiento de ambas herramientas, y terminaremos explicando cómo crear ejecutables .exe para Windows a partir de nuestro programa en Python.

distutils

Todo programa distribuido con `distutils` contiene un script llamado por convención setup.py, que se encarga de instalar la aplicación llamando a la función `setup` de `distutils.core`. Esta función tiene montones de argumentos, que controlan, entre otras cosas, cómo instalar la aplicación.

Destinados a describir la aplicación tenemos los siguientes argumentos:

- `name`: El nombre del paquete.
- `version`: El número de versión.
- `description`: Una línea describiendo el paquete.
- `long_description`: Descripción completa del paquete.
- `author`: Nombre del autor de la aplicación.
- `author_email`: Correo electrónico del autor.
- `maintainer`: Nombre de la persona encargada de mantener el paquete, si difiere del autor.
- `maintainer_email`: Correo de la persona encargada de mantener el paquete, si difiere del autor.
- `url`: Web de la aplicación.
- `download_url`: Url de la que descargar la aplicación.
- `license`: Licencia de la aplicación

También tenemos argumentos que controlan los archivos y directorios que deben instalarse, como son `packages`, `py_modules`, `scripts` y `ext_modules`.

El parámetro `scripts`, que es una lista de cadenas, indica el nombre del módulo o módulos principales, es decir, los que ejecuta el usuario. Si nuestra aplicación consistiera, por ejemplo, en un solo script ejemplo.py, el código de setup.py podría tener un aspecto similar al siguiente:

```
from distutils.core import setup

setup(name=”Aplicacion de ejemplo”,
  version=”0.1”,
  description=”Ejemplo del funcionamiento de distutils”, author=”Raul Gonzalez”,
  author_email=”zootropo en gmail”,
  url=”http://mundogeek.net/tutorial-python/”,
  license=”GPL”,
  scripts=[“ejemplo.py”]
)
```

Si hemos escrito otros módulos para ser utilizados por el script principal, estos se indican mediante el parámetro `py_modules`. Por ejemplo, supongamos que la aplicación consiste en un script principal ejemplo.py, y un módulo de apoyo

apoyo.py:

```
from distutils.core import setup

setup(name="Aplicacion de ejemplo",
  version="0.1",
  description="Ejemplo del funcionamiento de distutils", author="Raul Gonzalez",
  author_email="zootropo en gmail",
  url="http://mundogeek.net/tutorial-python/",
  license="GPL",
  scripts=["ejemplo.py"],
  py_modules=["apoyo"]
)
```

Para instalar paquetes Python (directorios que contienen varios módulos y un archivo __init__.py) usaríamos el parámetro `packages`. Si además del módulo ejemplo.py quisiéramos instalar los paquetes `gui` y `bbdd`, por ejemplo, haríamos algo así:

```
from distutils.core import setup

setup(name="Aplicacion de ejemplo",
  version="0.1",
  description="Ejemplo del funcionamiento de distutils", author="Raul Gonzalez",
  author_email="zootropo en gmail",
  url="http://mundogeek.net/tutorial-python/",
  license="GPL",
  scripts=["ejemplo.py"],
  packages=["gui", "bbdd"]
)
```

`ext_modules`, por último, sirve para incluir extensiones que utilice el programa, en C, C++, Fortran, …

Veamos ahora cómo se utilizaría el archivo setup.py una vez creado.
Al ejecutar el comando

```
python setup.py install
```

los módulos y paquetes especificados por `py_modules` y `packages` se instalan en el directorio Lib de Python. Los programas indicados en `scripts`, se copian al directorio Scripts de Python.

Una vez hemos comprobado que la aplicación se instala correctamente, procedemos a crear archivos mediante los que distribuir la aplicación a los usuarios. Para crear archivos con el código fuente se utiliza la opción `sdist` de setup.py, que crea por defecto un archivo tar.gz en Unix y un zip en Windows.

```
python setup.py sdist
```

Sin embargo se puede utilizar `--formats` para especificar el formato o formatos que queramos generar

bztar	.tar.bz2
gztar	.tar.gz
tar	.tar
zip	.zip

ztar	.tar.Z

Para crear un archivo tar.bz2, un tar.gz y un zip, por ejemplo, se utilizaría la siguiente orden:

```
python setup.py sdist --formats=bztar,gztar,zip
```

Para generar un archivo de distribución binaria, se usa la opción `bdist`:

```
python setup.py bdist
```

Los formatos que soporta `bdist` son los siguientes:

rpm	RPM
gztar	.tar.gz
bztar	.tar.bz2
ztar	.tar.Z
tar	.tar
wininst	Instalador Windows
zip	.zip

Para crear un archivo rpm y un instalador de Windows, por ejemplo, escribiríamos:

```
python setup.py bdist --formats=wininst,rpm
```

También es posible crear otros tipos de archivos de distribución utilizando scripts que extienden distutils, como es el caso de los paquetes deb mediante el script stdeb (http://stdeb.python-hosting.com/)

setuptools

setuptools extiende distutils añadiendo una serie de funcionalidades muy interesantes: introduce un nuevo formato de archivo para distribución de aplicaciones Python llamado *egg*, se encarga de buscar todos los paquetes que deben instalarse y añadir las posibles dependencias, permite instalar paquetes de PyPI con un solo comando, etc.

Además, como setuptools se basa en distutils, un script de instalación básico utilizando setuptools es prácticamente igual a su equivalente con distutils. Tan sólo cambiaría la sentencia de importación.

```
from setuptools import setup

setup(name=”Aplicacion de ejemplo”,
  version=”0.1”,
  description=”Ejemplo del funcionamiento de distutils”, author=”Raul Gonzalez”,
  author_email=”zootropo en gmail”,
  url=”http://mundogeek.net/tutorial-python/”,
  license=”GPL”,
  scripts=[“ejemplo.py”],
)
```

El único inconveniente que podríamos encontrar al uso de setuptools es que no está incluido por defecto en Python 2.5, aunque es probable que esto cambie en próximas versiones debido a su gran uso. Pero los desarrolladores de setuptools han pensado en todo, e incluso esto no debería suponer ningún problema, ya que con un mínimo esfuerzo por nuestra parte podemos hacer que setuptools se descargue e instale automáticamente en la máquina del usuario si este no se encuentra ya en el sistema. Basta distribuir con nuestro paquete un pequeño módulo extra ez_setup.py que viene incluido por defecto con setuptools (http://peak.telecommunity.com/dist/ez_setup.py) y llamar a la función `use_setuptools` del módulo al inicio de setup.py:

```
from ez_setup import use_setuptools
use_setuptools()

from setuptools import setup

setup(name=”Aplicacion de ejemplo”,
  version=”0.1”,
  description=”Ejemplo del funcionamiento de distutils”, author=”Raul Gonzalez”,
  author_email=”zootropo en gmail”,
  url=”http://mundogeek.net/tutorial-python/”,
  license=”GPL”,
  scripts=[“ejemplo.py”],
)
```

Veamos ahora con más detenimiento algunos de los cambios y novedades que introduce setuptools.

Integración con PyPI

Al estilo de CPAN en Perl setuptools permite instalar de forma fácil y sencilla los

paquetes pertenecientes a PyPI, el Índice de Paquetes Python (http://pypi.python.org/pypi), así como subir nuestros propios paquetes.

PyPI cuenta en el momento de escribir estas líneas con 4782 paquetes, por lo que poder instalar los paquetes de este repositorio con un simple comando supone una ayuda muy a tener en cuenta.

Instalar un paquete de PyPI es tan sencillo como pasar al comando `easy_install` el nombre del paquete a instalar

```
easy_install docutils
Searching for docutils
Reading http://pypi.python.org/simple/docutils/
Reading http://docutils.sourceforge.net/
Best match: docutils 0.5
Downloading http://prdownloads.sourceforge.net/docutils/
docutils-0.5.tar.gz?download
Processing docutils-0.5.tar.gz
Running docutils-0.5/setup.py -q bdist_egg --dist-dir /tmp/easy_install-wUAyUZ/docutils-
0.5/egg-dist-tmp-kWkkkv
"optparse" module already present; ignoring extras/optparse.py.
"textwrap" module already present; ignoring extras/textwrap.py.
zip_safe flag not set; analyzing archive contents…
docutils.writers.newlatex2e.__init__: module references __file__
docutils.writers.pep_html.__init__: module references __file__
docutils.writers.html4css1.__init__: module references __file__
docutils.writers.s5_html.__init__: module references __file__
docutils.parsers.rst.directives.misc: module references __file__
Adding docutils 0.5 to easy-install.pth file
Installing rst2pseudoxml.py script to /usr/bin
Installing rst2html.py script to /usr/bin
Installing rst2latex.py script to /usr/bin
Installing rst2s5.py script to /usr/bin
Installing rst2newlatex.py script to /usr/bin
Installing rstpep2html.py script to /usr/bin
Installing rst2xml.py script to /usr/bin
Installed /usr/lib/python2.5/site-packages/docutils-0.5-py2.5.egg
Processing dependencies for docutils
Finished processing dependencies for docutils
```

Poder subir nuestros paquetes a PyPI requiere de un proceso un poco más laborioso. Primero registramos los detalles de nuestra aplicación en PyPI mediante la opción register del script setup.py, el cual nos preguntará por nuestro nombre de usuario, contraseña y correo electrónico si no tenemos cuenta en PyPI, o nombre de usuario y contraseña si nos registramos anteriormente:

```
python setup.py register
running register
running egg_info
creating Aplicacion_de_ejemplo.egg-info
writing Aplicacion_de_ejemplo.egg-info/PKG-INFO
writing top-level names to Aplicacion_de_ejemplo.egg-info/top_level.txt
writing dependency_links to Aplicacion_de_ejemplo.egg-info/dependency_links.txt
writing manifest file 'Aplicacion_de_ejemplo.egg-info/SOURCES.txt'
reading manifest file 'Aplicacion_de_ejemplo.egg-info/SOURCES.txt'
writing manifest file 'Aplicacion_de_ejemplo.egg-info/SOURCES.txt'
We need to know who you are, so please choose either:
1. use your existing login,
2. register as a new user,
3. have the server generate a new password for you (and email it to you), or
4. quit
Your selection [default 1]: 1
```

```
Username: zootropo
Password:
Server response (200): OK
I can store your PyPI login so future submissions will be faster.
(the login will be stored in /home/zootropo/.pypirc)
Save your login (y/N)?y
```

Para crear y subir una distribución con el código fuente de nuestra aplicación se utiliza la opción `sdist upload`:

```
python setup.py sdist upload
```

También podríamos crear y subir un egg (un formato de archivo para distribuir aplicaciones Python que veremos en la próxima sección) utilizando la opción `bdist_egg upload`:

```
python setup.py bdist_egg upload
```

O combinar los tres pasos en un solo comando:

```
python setup.py register sdist bdist_egg upload
```

Una vez subido el paquete cualquier persona podría instalarlo en su sistema utilizando `easy_install`, de la misma forma que cualquier otro paquete de PyPI:

```
easy_install mi-paquete
```

Eggs

Los *eggs* (huevo en inglés) son archivos de extensión .egg mediante los que distribuir aplicaciones en Python. Serían algo así como el equivalente a los archivos .jar del mundo Java. Son multiplataforma, permiten manejar dependencias, y permiten instalar distintas versiones del mismo paquete.

La forma más sencilla de instalar aplicaciones distribuidas como archivos egg es mediante el comando easy_install, el cual comentamos brevemente en el punto anterior al hablar sobre su uso para instalar paquetes de PyPI. Para instalar un archivo egg no tenemos más que pasarle el nombre del archivo al comando easy_install:

```
easy_install mi-aplicacion.egg
```

o bien podemos pasarle la URL de la que descargar el egg:

```
easy_install http://mundogeek.net/mi-aplicacion.egg
```

Para construir nuestros propios eggs podemos utilizar el comando `bdist_egg` de setup.py, de forma similar a la manera en que construíamos paquetes RPM o instaladores para Windows con distutils:

```
python setup.py bdist_egg
```

Otros cambios destacables

Uno de los cambios más interesantes es la incorporación de un nuevo argumento para la función setup llamado `install_requires`, que consiste en una cadena o lista de

cadenas que indica los paquetes de los que depende la aplicación. Si nuestra aplicación necesitara tener instalado el paquete apoyo para poder ejecutarse, por ejemplo, escribiríamos lo siguiente:

```
install_requires = [“apoyo”]
```

Y de esta forma, easy_install se encargaría de buscar e instalar el paquete si fuera necesario, bien en PyPI, o en cualquier otro repositorio indicado por el parámetro `dependency_links`.

Además podemos especificar que se necesita una versión concreta del paquete requerido, que sea mayor o menor que una cierta versión, o que no se trate de una versión determinada utilizando operadores relacionales (==, `!=`, <, <=, >, >=):

```
install_requires = [“apoyo >= 1.0 < 2.0”]
```

También existen argumentos similares para declarar paquetes que deben instalarse para poder ejecutar el script de instalación (`setup_requires`), para poder ejecutar las posibles pruebas incluidas con el paquete (`tests_require`) y para conseguir funcionalidades adicionales (`extras_require`, que consiste en este caso en un diccionario).

setuptools incluye también atajos útiles, como la función `find_packages()` que nos evita tener que listar todos y cada uno de los paquetes que utiliza nuestro script en el parámetro `packages`, como era el caso de distutils:

```
from ez_setup import use_setuptools
use_setuptools()

from setuptools import setup, find_packages

setup(name=”Aplicacion de ejemplo”,
  version=”0.1”,
  description=”Ejemplo del funcionamiento de distutils”,
  author=”Raul Gonzalez”,
  author_email=”zootropo en gmail”,
  url=”http://mundogeek.net/tutorial-python/”,
  license=”GPL”,
  scripts=[“ejemplo.py”],
  packages = find_packages()
)
```

Crear ejecutables .exe

Tanto en Mac OS como en la mayor parte de las distribuciones Linux el intérprete de Python está instalado por defecto, por lo que los usuarios de estos sistemas no tienen mayor complicación a la hora de instalar y ejecutar aplicaciones escritas en Python.

En el caso de Windows, esto no es así, por lo que sería interesante que los usuarios de este sistema operativo no tuvieran que instalar el intérprete de Python. También sería interesante que nuestro programa consistiera en un archivo .exe en lugar de uno o varios archivos .py, para simplificar las cosas.

Todo esto lo podemos lograr gracias a py2exe, una extensión para distutils que, como su nombre indica, permite crear ejecutables para Windows a partir de código Python, y que permite ejecutar estas aplicaciones sin necesidad de tener instalado el intérprete de Python en el sistema.

Py2exe funciona examinando nuestro código fuente en busca de los módulos y paquetes que utilizamos, compilándolos y construyendo un nuevo archivo que incluye estos archivos y un pequeño intérprete de Python integrado.

Para probar el funcionamiento de py2exe creemos un pequeño programa ejemplo.py

```
print “Soy un .exe”
```

y el archivo setup.py correspondiente. Los cambios que tenemos que realizar a setup.py son sencillos: importar py2exe, y utilizar los argumentos `console` y `windows` para indicar el nombre del script o scripts que queramos convertir en ejecutables de consola o ejecutables de interfaz gráfica, respectivamente.

```
from distutils.core import setup
import py2exe

setup(name=”Aplicacion de ejemplo”,
  version=”0.1”,
  description=”Ejemplo del funcionamiento de distutils”,
  author=”Raul Gonzalez”,
  author_email=”zootropo en gmail”,
  url=”http://mundogeek.net/tutorial-python/”,
  license=”GPL”,
  scripts=[“ejemplo.py”],
  console=[“ejemplo.py”]
)
```

Para crear el ejecutable, utilizamos una nueva opción de línea de comandos para setup.py disponible tras importar el módulo y llamada, cómo no, py2exe:

```
python setup.py py2exe
```

Con esto py2exe generará un directorio build, con las librerías compiladas, y un directorio dist, con los archivos que conforman nuestra aplicación.
Entre los archivos que podemos encontrar en dist tendremos uno o varios ejecutables con el mismo nombre que los scripts indicados en `console` y `windows`, un archivo

python*.dll, que es el intérprete de Python, y un archivo library.zip, que contiene varios archivos pyc que son los módulos que utiliza la aplicación compilados.
Si queremos reducir el número de archivos a distribuir, podemos utilizar la opción `--bundle` de py2exe para añadir a library.zip las dll y los pyd (`--bundle 2`) o las dll, los pyd y el intérprete (`--bundle 1`).

```
python setup.py py2exe --bundle 1
```

o bien podemos añadir un nuevo argumento options a la función setup que indique el valor a utilizar (opción `bundle_files`), de forma que no tengamos que añadir el flag `--bundle` cada vez que usemos el comando py2exe:

```
from distutils.core import setup
import py2exe

setup(name=”Aplicacion de ejemplo”,
  version=”0.1”,
  description=”Ejemplo del funcionamiento de distutils”,
  author=”Raul Gonzalez”,
  author_email=”zootropo en gmail”,
  url=”http://mundogeek.net/tutorial-python/”,
  license=”GPL”,
  scripts=[“ejemplo.py”],
  console=[“ejemplo.py”],
  options={“py2exe”: {“bundle_files”: 1}}
)
```

Por último podemos incluso prescindir de library.zip e incrustarlo en el ejecutable utilizando el argumento `zipfile=None`

```
from distutils.core import setup
import py2exe

setup(name=”Aplicacion de ejemplo”,
  version=”0.1”,
  description=”Ejemplo del funcionamiento de distutils”,
  author=”Raul Gonzalez”,
  author_email=”zootropo en gmail”,
  url=”http://mundogeek.net/tutorial-python/”,
  license=”GPL”,
  scripts=[“ejemplo.py”],
  console=[“ejemplo.py”],
  options={“py2exe”: {“bundle_files”: 1}},
  zipfile=None
)
```

Índice analítico

Símbolos

A

B

C

D

E

F

N

O

P

R

S



Raúl González Duque es un Ingeniero Informático aficionado a los gadgets, la programación, Internet, la ciencia ficción, la tecnología, y todo lo relacionado con el mundo geek. Nacido en Toledo, lleva toda la vida viviendo en Madrid, desde donde se dedica a desarrollar aplicaciones.